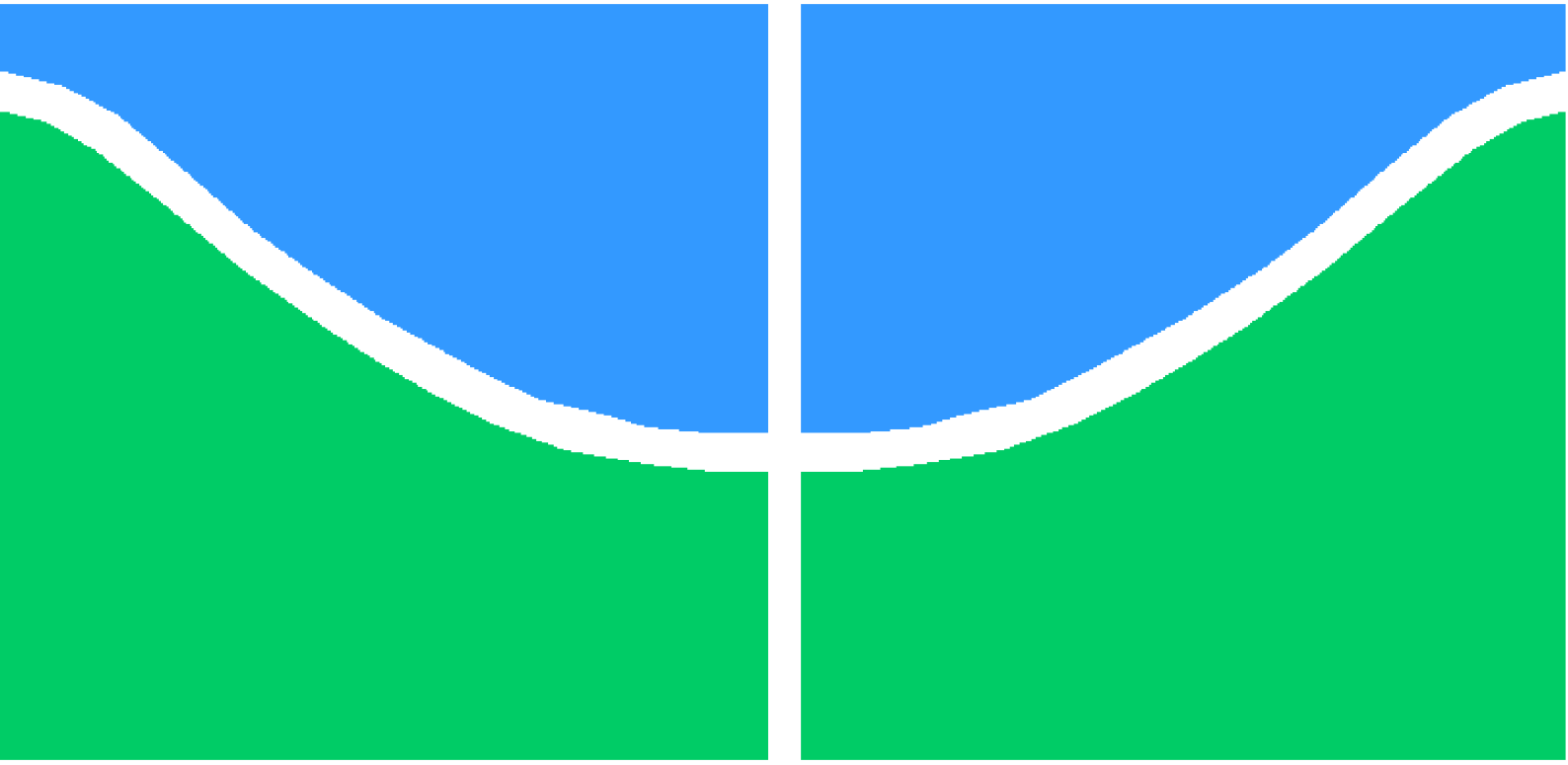


Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Engenharia de Software

Monitoramento e DevSecOps em Arquitetura Orientada a Eventos: Um estudo de caso com microsserviços e funções *serverless* em OpenFaaS

Autor: João Pedro Sconetto

Orientador: Professora Doutora Carla Silva Rocha Aguiar

Brasília, DF

2020

João Pedro Sconetto

Monitoramento e DevSecOps em Arquitetura Orientada a Eventos: Um estudo de caso com microsserviços e funções *serverless* em OpenFaaS

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Professora Doutora Carla Silva Rocha Aguiar

Brasília, DF

2020

João Pedro Sconetto

Monitoramento e DevSecOps em Arquitetura Orientada a Eventos: Um estudo de caso com microsserviços e funções *serverless* em OpenFaaS/ João Pedro Sconetto. – Brasília, DF, 2020-

94 p. : il. (algumas color.) ; 30 cm.

Orientador: Professora Doutora Carla Silva Rocha Aguiar

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2020.

1. Microsserviços. 2. DevSecOps. 3. *Serverless*. I. Professora Doutora Carla Silva Rocha Aguiar. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Monitoramento e DevSecOps em Arquitetura Orientada a Eventos: Um estudo de caso com microsserviços e funções *serverless* em OpenFaaS

CDU 00:000:000.0

João Pedro Sconetto

Monitoramento e DevSecOps em Arquitetura Orientada a Eventos: Um estudo de caso com microsserviços e funções *serverless* em OpenFaaS

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, – de – de 2020:

Professora Doutora Carla Silva Rocha Aguiar
Orientador

Professora Doutora Milene Serrano
Convidado 1

Professor Doutor Tiago Alves da Fonseca
Convidado 2

Brasília, DF
2020

Este trabalho é dedicado a todos sonhadores incansáveis,
que veem nas menores oportunidades grandes chances de realizar
o inimaginável.

Agradecimentos

Gostaria de agradecer imensamente a minha família, eles são e têm sido meu sustentáculo durante toda essa fase e toda esta vida. Agradeço ao meu pai, Pedro, pelos ensinamentos, pela curiosidade e pelo companheirismo, grande parte do que eu sou hoje se deve a ele. Agradeço a minha mãe, Cibele, por sempre estar ao meu lado, por ser minha melhor amiga, por ter paciência e me instruir a importância do caráter e do conhecimento. Às minhas irmãs, Isabella e Raphaella, agradeço por estarem ao meu lado e por terem me dado forças. Cada uma de seu modo me ajudou e têm sido fonte de inspiração para que eu conclua minha graduação.

Tenho gratidão à minha namorada, Mariana, ela que é minha companheira de graduação e de vida. Agradeço por estar ao meu lado, me amparar em todos os momentos que passamos juntos e ter me dado ânimo para continuar. Seu entusiasmo me faz querer ser melhor e ir mais longe, por isto, serei eternamente grato.

Por fim, gostaria de deixar registrado minha gratidão com todos que, direta ou indiretamente, fizeram parte desta história, pois sei que cada segundo compartilhado com cada um de todos me ajudaram a me tornar uma pessoa melhor. Em especial, um agradecimento a minha orientadora, Carla, por ter acreditado na minha ideia e tornar esse trabalho possível.

"Para todo problema complexo existe sempre uma solução simples, elegante e completamente errada."
H. L. Mencken

Resumo

A crescente demanda por produtos de software, aliado à inclusão digital e ao avanço de diversas áreas da tecnologia, impactou no formato como serviços são construídos e disponibilizados para os usuários. Para tentar acompanhar esta demanda, novos formatos arquiteturais surgiram e alguns outros evoluíram. Ao mesmo tempo, técnicas, práticas e metodologias também emergiram para sanar pontos fracos e agilizar o ciclo de vida do desenvolvimento de software. Este trabalho busca, por meio de um estudo de caso, a implementação de um produto (o Project SRC), juntamente a uma *stack* de serviços, com o uso de funções como serviços de software (*FaaS*) e microsserviços, aliados a automações e práticas *DevOps*. Relatando, o ciclo de desenvolvimento do projeto, desde o levantamento dos requisitos até monitoramento em uso, identificando particularidades e lições aprendidas no desenvolvimento de uma solução utilizando deste formato.

Palavras-chave: Microsserviços. Arquitetura Orientada a Eventos. *DevSecOps*. *Serverless*. *FaaS*. *OpenFaaS*.

Abstract

The growing demand for software products, coupled with digital inclusion and the advancement of various areas of technology, has impacted on the way services are built and made available to users. To try to keep up with this demand, new architectural formats have emerged and some others have evolved. At the same time, techniques, practices and methodologies have also surfaced to address weaknesses and streamline the software development life cycle. This work seeks, through a case study, to report the development of a software product (the Project SRC), with a service stack, using an architectural format arising from the need to build software more quickly, the microservices and the functions as a software service (FaaS), combined with the use of DevOps practices and techniques, that seeks to speed up the integration and delivery of software. Presenting, the project development cycle, from the requirements phase to software deployment and usage monitoring, identifying particularities and lessons learned in the development of a solution using this architecture.

Key-words: Microservices. Event-driven Architecture. DevSecOps. Serverless. FaaS. OpenFaaS.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
APS	*Application Service Providing*
BD	Banco de Dados
CAMS	*Culture, Automation, Measurement and Sharing*
CD	*Continuous Deployment*
CI	*Continuous Integration*
DNS	*Domain Name System*
FaaS	*Function as a Service*
HTTP	*Hypertext Transfer Protocol*
IaaS	*Infrastructure as a Service*
JSON	*JavaScript Object Notation*
LXC	*LinuX Container*
MQ	*Message Queue*
PaaS	*Platform as a Service*
P2P	*Peer-to-Peer*
REST	*Representational State Transfer*
RPC	*Remote Procedure Call*
SaaS	*Software as a Service*
SOA	*Service-Oriented Architecture*
SSH	*Secure Shell*
TI	Tecnologia da Informação
UnB	Universidade de Brasília
VM	*Virtual Machine*
XML	*Extensible Markup Language*

Sumário

1 Introdução

Com o rápido avanço das tecnologias, hoje, temos mais contato com software do que tínhamos antes da popularização dos computadores pessoais. De acordo com a Nações Unidas (United Nations, 2018), no final do ano de 2018, cerca de 51,2% da população mundial (3,9 bilhões de pessoas) já possuía acesso à internet e estão *online*. Essa revolução digital gerou crescentes demandas de produtos de software complexos.

Logo, o formato arquitetural dos produtos de software construídos precisaram se adaptar a essas necessidades. Arquiteturas monolíticas ou de camada única começaram a ser substituídas por arquiteturas de N camadas, orientadas a eventos, microsserviços, microkernel, entre outras. As evoluções buscavam tratar o tamanho, a complexidade e o desempenho para responder a demanda da produção.

Dentro dessas novas arquiteturas que foram sendo elaboradas e melhoradas a partir de outras arquiteturas já existentes, é válido citar a arquitetura que ficou conhecida com SOA, da sigla em inglês *Service-Oriented Architecture*, ou, arquiteutra orientada a serviços. Trata-se de um formato de arquitetura de software que (HE, 2003) busca alcançar o baixo acoplamento entre agentes de software interativos. Um serviço, então, é definido como uma unidade de trabalho feito por um provedor de serviços para alcançar os resultados finais desejados por um consumidor deste serviço. Este padrão é comumente usado para construir serviços web, com RPC (*Remote Procedure Call*) ou REST (*Representational State Transfer*).

Surge, posteriormente, outro formato de arquitetura, o de microsserviços, que contém algumas semelhanças com o SOA, mas com alguns princípios diferentes. Fowler e Lewis (FOWLER; LEWIS, 2015) descrevem e tentam definir este estilo arquitetural. Os autores buscam definir o termo, que à época ainda estava evoluindo e tomando popularidade. Temos que “Enquanto não há uma definição precisa deste estilo arquitetural, há certas características em comum acerca da sua organização sobre as capacidades de negócio, implantação automatizada, inteligência nos pontos terminais, e controle descentralizado das linguagens e dos dado”. Podemos dizer então que o estilo arquitetural de microsserviços é uma abordagem para desenvolver aplicações a partir de um conjunto de pequenos serviços.

Seguindo a ideia de decomposição de serviços e funcionalidades de lógicas de negócio em unidades menores, surge outra evolução, as funções como serviço, ou, *FaaS* (*Function as a Service*). Também conhecida como *serverless*, que envolve outra característica deste estilo de implementação de serviços, trata de um formato arquitetural onde os serviços são fragmentados numa granularidade tão baixa, que o serviço se torna uma

função especialista. Para este termo não há uma definição formal, e algumas concepções que diferem, mas há na comunidade (FOX et al., 2017) levantamentos da extensão do que pode ser dito como uma função como um serviço.

Com o advento destes novos modelos arquiteturais de construção de software, fases do ciclo de vida como implantação, integração, controle, entre outras, precisaram ser reforçadas e evoluídas de forma a acompanhar e gerenciar todos os diferentes serviços. Há estudos (BALALAIE; HEYDARNOORI; JAMSHIDI, 2016) que afirmam que estes formatos de arquitetura habilitam os princípios pregados pelo *DevOps*. Logo, a cultura das práticas do *DevOps*, que buscam exatamente unificar, automatizar e manter o desempenho (LEITE et al., 2019) as fases acima citadas, foram de encontro com os estilos arquiteturais. Mas, devido a relativa contemporaneidade dos formatos arquiteturais, principalmente o das funções como serviço, não há padrões e melhores práticas largamente definidas pela comunidade de software de como garantir a cultura do *DevOps*.

Neste trabalho, é apresentado um relato de experiência, de como aplicar as práticas do *DevOps* em arquiteturas microsserviços e funções como serviços, de maneira que formatos arquiteturais similares consigam facilmente implementar a cultura em seus processos durante o ciclo de vida do software. Com auxílio de técnicas de medição, monitoramento e segurança são levantados os desafios e os atalhos para que o controle de uma arquitetura orientada a eventos a partir de serviços e funções, dentro das práticas necessárias, não seja oneroso ao ciclo do produto. Ao final construiu-se uma proposta de melhores práticas a partir das experiências.

1.1 Objetivo Geral

O objetivo principal que rege as construções propostas neste estudo é a elaboração de orientações e melhores práticas da aplicação da cultura do *DevOps* em arquiteturas microsserviços e funções como serviço.

1.1.1 Objetivos específicos

Para alcançar tal objetivo, será mandatório:

- Conceber e construir a arquitetura de um conjunto de microsserviços e funções *FaaS*;
- Conceber e construir um protótipo de prova de conceito para aplicação Project SRC;
- Conceber e alocar a infraestrutura para implantação da arquitetura (aplicações) e das práticas de *DevOps*;
- Aplicar e monitorar o processo das práticas de *DevOps* e segurança na arquitetura proposta do trabalho;

- Levantar pontos fortes e fracos da arquitetura em todos os processos, e;
- Construir a proposta com melhores práticas.

1.2 Estrutura do Trabalho

Este trabalho está organizado no seguinte arranjo. A Fundamentação Teórica dá uma perspectiva inicial aos principais temas, tecnologias e informações que se julgou necessário entendimento antes da execução e construção da solução proposta. Ainda na Fundamentação, temos algumas seções com as ferramentas usadas, onde há a explicação técnica de algumas das principais ferramentas usadas para a construção do projeto, atentando-se para as tecnologias primordiais do trabalho. Na Proposta, é elucidado com mais detalhes qual o problema levantado, como se propõe resolvê-lo e como está divido logicamente o formato da solução. Apresenta também também a metodologia, onde é apresentado como se deu a execução do trabalho, apresentando informações sobre as métodos, processos, monitoramento e os demais artifícios usados para a formalização do processo de trabalho, com as verificações exigidas. Por fim, no Projeto, faz-se apresentação e o esclarecimento dos resultados, o processo e o percurso da implementação, o relato das lições aprendidas perante a execução do trabalho, e, por fim, a conclusão deste trabalho.

2 Referencial

Neste capítulo, está presente o estudo prévio e um levantamento referencial dos principais conceitos e tecnologias que cerceiam a construção deste trabalho. Será apresentado, de cada um dos temas centrais, a definição necessária para o desenvolvimento do projeto, dentro deste âmbito e que se julgou indispensável. Desta forma, passos posteriores serão baseados nas concepções teóricas aqui descritas.

2.1 *DevOps*

DevOps é (BASS; WEBER; ZHU, 2015; ZHU; BASS; CHAMPLIN-SCHARFF, 2016) um conjunto de práticas destinadas a diminuir o tempo entre a publicação de uma alteração em um sistema e a mudança sendo aplicada na produção normal, enquanto se garante alta qualidade. Podemos definir o *DevOps*, em outras palavras, como sendo (LEITE et al., 2019) um esforço colaborativo e multidisciplinar dentro de uma organização para automatizar a entrega contínua de novas versões de software, garantindo sua corretude e confiabilidade. Logo, com essas duas definições, podemos inferir que o *DevOps* é uma cultura que busca automatizar tarefas de desenvolvimento e operações, garantir qualidade, confiabilidade e o mínimo de erros durante os processos automatizados.

Sua extensão, *DevSecOps*, é (MYRBAKKEN; COLOMO-PALACIOS, 2017) a expansão necessária do *DevOps*, onde o propósito é integrar controles de segurança e processos seguros no ciclo de desenvolvimento do software, e no *DevOps*, que é alcançado pela promoção da colaboração entre os times de segurança, desenvolvimento e operações.

2.1.1 DevSecOps

Os princípios que baseiam o *DevSecOps* são comuns aos do *DevOps* e ao aos princípios CAMS [1] (*Culture*, *Automation*, *Measurement* e *Sharing* – Cultura, Automação, Medição e Compartilhamento), com a adição da segurança a partir do início do ciclo de vida do produto, como pode ser visto na Tabela 1.

Portanto, busca trazer mais à esquerda o processo de segurança dentro do ciclo de vida do software. Em processos de desenvolvimentos tradicionais, a segurança é um passo perto do final do processo. Com o *DevSecOps*, promove-se esse deslocamento à esquerda (do ponto de vista do processo), onde ele é incluído em cada parte do processo de desenvolvimento. Isso significa que os times de segurança são envolvidos desde o primeiro

[1] Modelo CAMS criado por Damon Edwards e John Willis, com princípios de *DevOps.*, sendo constituídos pelo conjunto de valores usados pelos engenheiros de *DevOps*

Tabela 1 – Princípios de CAMS do *DevOps* aliados à segurança.

Princípios CAMS	
Cultura	A cultura *DevOps* promove a colaboração entre times de desenvolvimento e operação, onde ambos aceitam que são responsáveis pela entrega do software para o usuário final. O *DevSecOps* inclui a colaboração com o time de segurança, promovendo uma cultura, onde ambos times também trabalham para incluir segurança nas tarefas. Tal prática faz com que todos também pensem em questões de seguranças envolvidas em suas tarefas;
Automação	No *DevOps*, a automação da construção, implantação e teste é importante para alcançar desenvolvimento, entrega e *feedback* do usuário ágil. O *DevSecOps* promove a automação da segurança também, para acompanhar a escala similar ao *DevOps*. O objetivo deve ser a automação de todos os dispositivos de controle de segurança, onde estes possam ser implantados e gerenciados sem a interferência manual;
Medição	No *DevOps*, a medição incluem monitorar métricas de negócio e indicadores chave de performance, como estabilidade do sistema, para saber o estado atual do mesmo e descobrir como melhorá-lo. O *DevSecOps* promove o uso e desenvolvimento de métricas que mantenham registro de ameaças e vulnerabilidades durante todo o processo de desenvolvimento do software;
Compartilhamento	No *DevOps*, desenvolvedores e operadores compartilham conhecimento, ferramentas de desenvolvimento e técnicas para gerenciar o processo. O *DevSecOps* promove a inclusão do time de segurança no compartilhamento feito pelo *DevOps*. Deixar o time de segurança ciente dos desafios enfrentados pelos operadores e desenvolvedores, e vice versa, ajuda a melhorar e desenvolver o processo de segurança.

Fonte: Guthrie (2019)

passo de planejamento e fazem parte de cada iteração do planejamento do ciclo de desenvolvimento. O que também significa que a segurança está lá para ajudar desenvolvedores e operadores nas considerações de segurança.

2.1.2 NoOps

O *NoOps* (*no operations* – sem operações) é um conceito que um ambiente de TI pode se tornar tão automatizado e abstraído da infraestrutura subjacente que não há necessidade de uma equipe dedicada para gerenciar o software internamente (ROUSE, 2015). Em contraponto a um formato tradicional, onde após a entrega do software desenvolvido e testado, uma equipe de operações, a qual implanta e mantém o software, fica responsável a partir desse ponto. Em um ambiente *NoOps*, a manutenção e outras tarefas

executadas pela equipe de operações seriam automatizadas.

Os dois principais impulsionadores do *NoOps* são o aumento da automação de TI e a computação em nuvem. Ao seu extremo, uma organização *NoOps* é uma organização a qual não tem nenhum funcionário de operações, entretanto, vários outros sistemas podem, também, ser chamados de "*NoOps*". Por exemplo, provedores de plataformas como serviço (*PaaS* - Platform as a Service) como AppFrog e Heroku descrevem seus produtos como plataformas *NoOps*.

NoOps significa que desenvolvedores podem codificar e implantar um serviço, gerenciar e escalonar o seu código (FARROHA; FARROHA, 2014). Tarefas operacionais ficam delegadas a sistemas automatizados, como o *CloudFoundry*, que irão gerenciar a manutenção da aplicação ao invés de administradores de sistemas (*SysAdmins*).

2.1.3 GitOps

O *GitOps* é outra vertente de controle de operações do ciclo de software, apoiando-se no uso de ferramentas de controle de versionamento, neste caso o Git, para automatizar e gerenciar tarefas de operações. Em uma simples definição, o *GitOps* trata-se da metodologia que faz o uso de *Pull Requests*[2] do Git para gerenciar o provisionamento de infraestrutura e a implantação de software (RILEY, 2018).

O conceito do *GitOps* se originou na *Weaveworks* (RICHARDSON, 2017), onde os desenvolvedores descrevem como utilizaram o Git para criar uma "fonte única da verdade", ou seja, a ferramenta de versionamento é usada tanto para controlar versões no desenvolvimento como histórico, revisão em pares, reversões, entre outros.

O uso de *pull requests* para gerenciar a infraestrutura e suas tarefas trazem alguns benefícios, como:

- Uma única ferramenta e interface para controlar infraestrutura. Isso evita a necessidade de usar uma ferramenta diferente para controlar diferentes tipos de infraestrutura;
- Controle de versão para todas as alterações feitas na sua configuração. Isso é útil para reverter alterações, bem como para fins de auditoria;
- Poder usar a ferramenta que apresenta as diferenças entre versões para detectar alterações e gerar alertas automaticamente. Isso significa não apenas poder monitorar constantemente as alterações, mas também que, se as condições reais divergirem da maneira de como devem ser as configurações, o problema será detectado facilmente;

[2] Um *pull request* (solicitação de recebimento) é um método de envio de contribuições para um projeto de desenvolvimento aberto. Geralmente, é a maneira de enviar contribuições para um projeto usando um sistema de controle de versão distribuído (DVCS), como o Git (JOHNSON, 2013).

- Ao usar os *pull requests* do Git (algo com o qual a maioria dos desenvolvedores de software já conhece), não é necessário ensinar à equipe uma nova ferramenta para gerenciar a infraestrutura.

Portanto, há razões que compelem o uso do Git e os *pull requests* como base do gerenciamento da infraestrutura. O *GitOps* ajuda os times a implementarem uma única prática básica, um conjunto de ferramentas e rastreamento de dados. Como resultado, reduz o número de variáveis no gerenciamento da infraestrutura, e fornece uma visibilidade vasta e contínua do estado da infraestrutura, facilitanto, também, a detecção de sinais de problemas e falhas quando eles surgem.

2.2 Microsserviços

A arquitetura microsserviços é (FOWLER; LEWIS, 2015) uma abordagem para desenvolver uma única aplicação como uma suíte de serviços, cada um sendo executado em seu próprio processo e se comunicando por mecanismos leves, usualmente por meio de uma *API HTTP*, como recurso desta interface. Estes serviços são construídos com base nos recursos de negócio, cada um com responsabilidades bem definidas e implementados de maneira independente por meio de implantações automatizadas. É importante ressaltar que deve haver um mínimo de gerenciamento centralizado desses serviços para seu funcionamento, que podem ser escritos em diferentes linguagens de programação e usar diferentes tecnologias.

Ao contrapor a arquitetura de microsserviços com o estilo monolítico, temos em contraste os dois formatos. Uma aplicação monolítica é construída como uma única unidade, que trata (abstraíndo como o monolito sendo a aplicação *server-side*) a requisição HTTP, executa a lógica do domínio, busca e atualiza os dados do banco de dados e, por fim, seleciona e popula as páginas HTML que serão servidas para o navegador. A aplicação é um único executável lógico. Qualquer mudança na aplicação envolve a construção e implantação de uma nova versão da aplicação. Neste formato, é possível escalar horizontalmente o monolito, rodando várias instâncias da aplicação por trás de um *load-balancer*.

Aplicações monolíticas podem ser bem sucedidas, mas existem restrições que normalmente afetam o ciclo de produto destes softwares. Os ciclos de mudanças são altamente amarrados – isto é, uma mudança feita em uma parte pequena da aplicação requer que o monolítico seja reconstruído e reimplantado inteiramente. Com o passsar do tempo, e a evolução da aplicação, muitas vezes torna-se difícil manter uma boa estrutura modular, dificultando a manutenção de alterações que devem afetar apenas um módulo específico. Escalar o monolito requer escalar a aplicação por inteiro ao invés de partes dele, demandando uma quantidade elevada de recursos (infraestrutura). Conforme exemplificado na Figura ??.

Essas restrições levaram ao estilo arquitetural microsserviços – construir aplicações como suítes de serviços. Aliado ao fato dos serviços serem escaláveis e implantáveis independentemente, cada serviço, também, provê uma fronteira delimitada do módulo, permitindo que diferentes serviços sejam escritos em linguagens diferentes, com tecnologias diferentes e, da mesma forma, gerenciados por times diferentes.

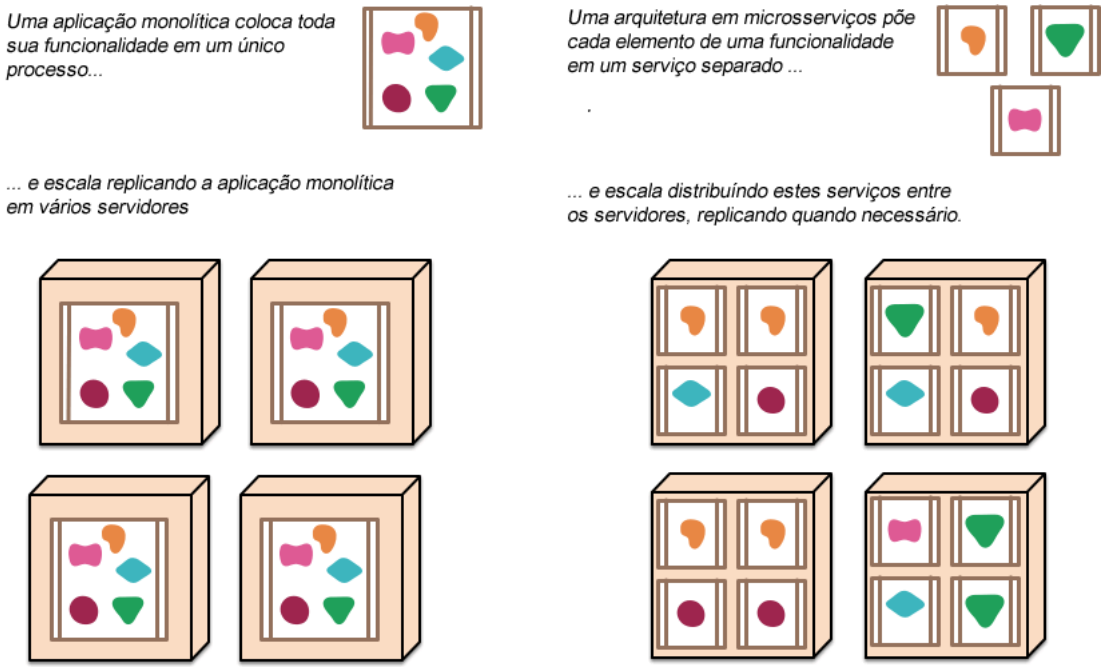


Figura 1 – Diferença entre sistemas monolíticos e microsserviços.
Fonte: Mendes (2016).

2.3 *Application Programming Interface*

Interface de programação de aplicações, do acrônimo em inglês API, trata-se de um conjunto de definições e protocolos usado no desenvolvimento e na integração de software de aplicações (RedHat Incorporated, 2019). Com as APIs, a solução/serviço pode se comunicar com outros produtos e serviços sem precisar saber como eles foram implementados, simplificando o desenvolvimento de aplicações que utilizam destes serviços. As APIs costumam ser vistas como contratos, com documentações que representam um acordo entre as partes interessadas. Se uma dessas partes enviar uma solicitação estruturada no formato específico presente na documentação, isso derteminará como o software da outra parte responderá.

Logo, as APIs são uma maneira simplificada de integrar novos componentes de uma aplicação à arquitetura pré-existente. Há também, dentro da categoria das APIs, as APIs chamadas remotas. Tratam-se de interfaces que foram projetadas para interagir por meio de uma rede de comunicações, e são comumente usadas na Web. Sendo assim, várias

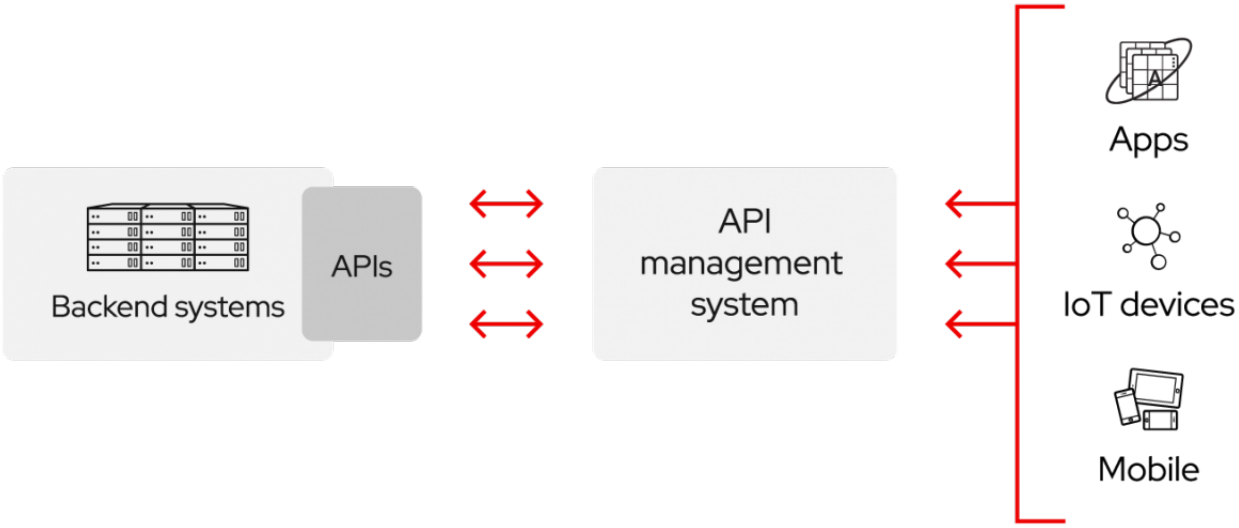


Figura 2 – Exemplo lógico de construção de API.
Fonte: RedHat Incorporated (2019).

APIs são projetadas com base em padrões da web. Nem todas as APIs remotas são Web. Contudo, é justo dizer que, em geral, as APIs Web são remotas.

As APIs web normalmente usam o protocolo HTTP (*Hypertext Transfer Protocol*) para mensagens de solicitação e fornecem uma definição da estrutura das mensagens de resposta. Essas mensagens de resposta geralmente têm o formato de arquivo XML (*Extensible Markup Language*) ou JSON (*JavaScript Object Notation*). Tanto XML quanto JSON são formatos de preferência, pois apresentam os dados de forma simplificada, o que facilita a manipulação por outras aplicações. As APIs web também podem ser divididas quanto ao seu princípio de operação e/ou restrições da sua concepção de funcionamento, podendo citar APIs que usam o princípio do RPC, ou então APIs REST, conforme levantado inicialmente no Capítulo 1.

2.4 *Cloud Computing*

O *cloud computing* (computação em nuvem) trata-se de uma abordagem de computação, envolve tanto armazenamento de dados, quanto processamento, implantação de serviços, entre outras atividades. Em vez dessas atividades serem desenvolvidas em *desktops* ou em servidores locais, elas passam a serem desenvolvidas e computadas na nuvem (HAYES, 2008). Outros nomes que referenciam o *cloud computing* são *on-demand computing* (computação sob demanda), *software as a service* (software como serviço) ou *internet as platform* (internet como plataforma), e o que todas têm em comum é a mudança "geográfica"da computação.

Sob guarda chuva do *cloud computing*, temos três dos principais componentes,

sendo eles: o *IaaS*, o *PaaS* e o *SaaS* (definidos nas Subseções 2.4.1, 2.4.2 e 2.4.3). Usualmente, aplicações usam do *IaaS* como infraestrutura. Este provê plataformas, *PaaS*, onde plataformas oferecem serviços de *SaaS*. Portanto, os três componentes se completam entre si e formam uma grande parcela dos serviços de *cloud computing*. Podemos ver, de forma simplista, como sendo uma camada de infraestrutura, acima uma camada de construção e por fim uma camada para consumo de software, como podemos ver na Figura ??.

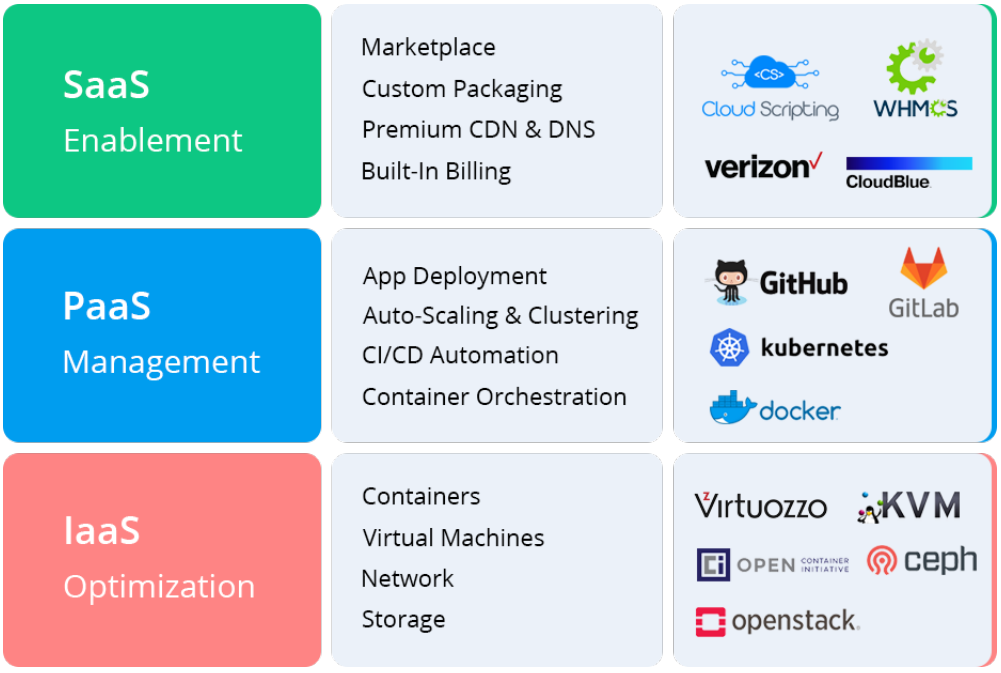


Figura 3 – *Cloud Computing* - *IaaS*, *PaaS* e *SaaS*.
Fonte: Fydorenchyk (2019).

Cada um destes três componentes é mais bem elucidado nas seções a seguir, informando de forma ampla sua natureza, concepção, serviços e provedores. Adicionalmente, são apresentados alguns dos pontos positivos e negativos do uso destes tipos de serviços de *cloud computing*.

2.4.1 *Infrastructure as a Service*

O IaaS, infraestrutura como serviço, é a entrega de hardware (servidor, armazenamento e rede), e software associado (tecnologia de virtualização de sistemas operacionais, sistemas de arquivos), como um serviço (BHARDWAJ; JAIN; JAIN, 2010). É uma evolução da hospedagem tradicional que não requer compromisso de longo prazo e permite que os usuários provisionem recursos sob demanda.

Diferentemente dos serviços PaaS, o provedor de IaaS faz pouco gerenciamento, focando em manter a infraestrutura operacional. Os usuários devem implantar e gerenciar os serviços de software da mesma maneira que fariam em sua própria infraestrutura. O

Amazon Web Services EC2 e o S3[3] são exemplos de ofertas de IaaS.

Infraestrutura como serviço é uma forma de hospedagem. Inclui acesso à rede, serviços de roteamento e armazenamento. O provedor de IaaS geralmente fornece os serviços administrativos e de *hardware* necessários para armazenar aplicativos e uma plataforma para aplicativos em execução. O dimensionamento da largura de banda, memória e armazenamento geralmente é incluído, e os fornecedores competem pelo desempenho e preços oferecidos em seus serviços dinâmicos. O provedor de serviços possui o equipamento e é responsável por abrigá-lo, executá-lo e mantê-lo. O IaaS pode ser adquirido com um contrato ou com o pagamento conforme o uso. No entanto, a maioria dos compradores considera o principal benefício da IaaS a flexibilidade dos preços, pois você só precisa pagar pelos recursos que a entrega de seus aplicativos exige (BHARDWAJ; JAIN; JAIN, 2010).

Características e componentes do IaaS incluem:

- Serviço de computação utilitário e modelo de cobrança;
- Automação de tarefas administrativas;
- Escala dinâmica;
- Virtualização do ambiente de trabalho;
- Serviços baseados em políticas, e;
- Conectividade à internet.

Logo, as IaaS fornecem serviços, a preços acordados, para manter uma infraestrutura externa ao dono do software que pode ser utilizada sem a preocupação com manutenção, provisionamento de *hardware*, armazenamento, configurações de rede, entre outros, cabendo apenas a implantação do software nesses serviços.

2.4.2 *Platform as a Service*

Enquanto o IaaS (Seção 2.4.1) fornece recursos de *hardware* sob demanda virtualmente, o SaaS são os aplicativos de software que o usuário usa pela rede. O conceito de plataforma como um serviço, PaaS, assume uma função intermediária – que geralmente não é visível – ao fornecer a plataforma operacional necessária para o software virtualmente oferecido. No entanto, o conceito PaaS também se tornou independente, como um serviço separado, como mostram os exemplos do Google App Engine. Assim, o PaaS está expandindo o modelo no ambiente SaaS com o provedor da plataforma como um ator

[3] Amazon Secure Storage Service. https://aws.amazon.com/pt/s3/. Acesso em 05/08/2020.

adicional, oferecendo assim seus próprios e interessantes novos aspectos para pesquisa e prática de informações de negócios (BEIMBORN; MILETZKI; WENZEL, 2011).

Portanto, podemos imaginar o PaaS como sendo um IaaS mais certa quantia de aplicações de software, como integrações com um conjunto comum de funções de programação ou bancos de dados como fundação, na qual se pode construir aplicações (BHARDWAJ; JAIN; JAIN, 2010).

O PaaS é definido, então, como (GROHMANN, 2009) o fornecimento de uma plataforma completa, ou seja, *hardware* e software, como um serviço, a fim de dar aos fabricantes de software independentes (os chamados ISVs, *Independent Software Vendors*) a oportunidade de desenvolver e operar soluções SaaS ou integrá-las a aplicativos de software tradicionais. A plataforma fornece aos ISVs todas as funcionalidades necessárias para o ciclo de vida, do desenvolvimento ao teste, à implantação e operação de aplicativos.

Conforme ocorre com outros formatos de serviço, tal prática permite um custo menor para manutenção das aplicações construídas pelos ISVs, tendo em vista os modelos de cobrança por ciclo ou por tempo de execução e retirando a preocupação de manter infraestrutura e outros componentes intermediários necessários para a execução da aplicação. Entretanto, o modelo de cobrança pode se tornar custoso quando a demanda por recursos é elevada, ou há necessidade de escalar vários serviços. Outras desvantagens, quando comparados com formatos tradicionais para plataforma de infraestrutura, incluem uma menor quantidade de recursos de sistema (recursos operacionais), controle reduzido dos recursos e dependência do gerenciamento de rede e tráfego do provedor.

Alguns provedores conhecidos de plataformas, já citados em outros tópicos, são o Amazon Web Services Elastic Beanstalk[4], Heroku[5], Apache Stratos[6], OpenShift[7], entre outros.

2.4.3 *Software as a Service*

O SaaS, software como serviço, é um formato de construção e comercialização de software. É um dos três principais componentes da computação em nuvem (*Cloud Computing*) (TSAI; BAI; HUANG, 2014), com os outros componentes sendo o PaaS – Seção 2.4.2 – e o IaaS – Seção 2.4.1. A ideia por trás do SaaS não é fundamentalmente nova. Já na década de 1990, uma abordagem semelhante foi usada sob o nome *Application Service Providing* (ASP), sendo o SaaS uma extensão do ASP.

Um destes formatos dá-se pelo uso do SaaS. Nesse caso, os consumidores recebem uma solução de software como um serviço via internet (BUXMANN; HESS; LEHMANN,

[4] AWS Elastic Beanstalk. https://aws.amazon.com/pt/elasticbeanstalk/. Acesso em 05/08/2020.
[5] Heroku. https://www.heroku.com/. Acesso em 05/08/2020.
[6] Apache Stratos. https://stratos.apache.org/. Acesso em 05/08/2020.
[7] OpenShift. https://www.openshift.com/. Acesso em 05/08/2020.

2008). O provedor do SaaS é responsável pela operação e manutenção do software multilocatário. Os fornecedores não ganham *royalties*. Em vez disso, os usuários pagam *royalties* pelos componentes e serviços de software alugados, geralmente mensalmente, trimestralmente ou anualmente. Além disso, da perspectiva de provedores de software e serviços, outros modelos de receita, tais como receita de publicidade ou faturamento com base no uso, é concebível.

Em outro formato, em nível de sistema, diferentemente de software tradicionais que são executados em sistemas operacionais, o SaaS é normalmente implantado em sistemas de plataforma como serviço, como o Google App Engine[8], Amazon EC2[9], ou Azure[10], ou outra estrutura especializada para SaaS.

O uso de SaaS traz alguns benefícios quando comparado ao formato normal de comercialização de software. O fornecedor fica a cargo de todo o suporte de TI pela utilização do SaaS, incluindo a manutenção diária, cópias de segurança dos dados, atualizações do software e segurança (DAN, 2007). Portanto, não se trata apenas do software, mas sim de toda a utilidade de computação junto ao produto. O que custeia o uso dessa solução, já que os usuários não precisam preocupar com a infraestrutura, o *hardware*, além de custos de treinamento.

2.4.4 *Function as a Service*

O FaaS, função como serviço, que conjuntamente leva o nome de *Serverless*, trata-se de um formato específico de fornecer serviços de software granulares e especialistas em funções específicas de domínios lógicos. O termo *Serverless* pode ser rastreado (FOX et al., 2017) no seu significado original de não haver servidores para o funcionamento do software, comum em aplicações P2P (*Peer-to-Peer*), onde a máquina do usuário funcionava como servidor e cliente. No contexto atual, de aplicações em nuvem, o termo diz respeito à não necessidade de preocupação do desenvolvedor em possuir um servidor (infraestrutura) para rodar as funções, e utilizar de serviços SaaS (*Software as a Service*, software como serviço) de grandes empresas como Google e Amazon, para hospedar essas funções. É importante citar que existem soluções nas quais é possível trazer para infraestruturas internas os requisitos para hospedar as funções. Normalmente, questões como essa envolvem o custo de manter em infraestrutura própria, contra o custo da utilização dos serviços SaaS dos provedores externos, bem como podendo sofrer influências de segurança, propriedade privada do software e outras questões.

Note que diferente de soluções SaaS e PaaS (*Platform as a Service*), que estão sempre em execução, mas escalam sob demanda, funções *serverless* têm seus processa-

[8] Google App Engine. https://cloud.google.com/appengine/. Acesso em 12/08/2020.
[9] Amazon Elastic Compute Cloud. https://aws.amazon.com/pt/ec2/. Acesso em 12/08/2020.
[10] Microsoft Azure. https://azure.microsoft.com/pt-br/. Acesso em 12/08/2020.

mentos executados sob demanda – apenas quando são invocados – e, consequentemente, escalados sob demanda.

Algumas definições de FaaS e *Serverless* levantadas no centro de pesquisa da IBM (Castro et al., 2017) são:

- Plataforma nativa para computação em nuvem (*cloud-native*);
- Computação sem estado *stateless*, de execução curta;
- Aplicações orientadas a eventos;
- Escalam e desescalam instantaneamente e automaticamente, e
- Cobrada pelo uso real em uma granularidade de milissegundos[11].

2.5 Mensageria (*Message Queue*)

Mensageria, ou *Message Queue – MQ* (fila de mensagem), trata-se de um formato usado na comunicação assíncrona entre sistemas (Amazon Web Services, Inc., 2019). Esse formato tem sem uso comumente aliado a arquiteturas sem servidor (*serverless)* e de microsserviços. As mensagens são armazenadas em uma fila até serem processadas e removidas. Cada mensagem é processada uma única vez, por um único consumidor (podendo haver aplicações onde mais de um consumidor processa mensagens). As filas de mensagens podem ser usadas para dissociar processamento pesado, para armazenar trabalho em *buffers* ou lotes, e para processar uniformemente picos de cargas de trabalho.

O formato, apesar de existir diversos outros protocolos proprietários como o JMS (*Java Message Service*) ou o MSMQ (*Microsoft Message Queuing*), o protocolo predominante das aplicações deste trabalho utiliza do AMQP (*Advanced Message Queuing Protocol*). O AMQP surgiu a partir da união de grandes empresas como a Red Hat, a Cisco Systems, a JPMorgan Chase, entre outras, formando o grupo AMQP, com o objetivo de criar um padrão aberto para um protocolo de mensagens assíncrono interoperável em escala corporativa (VINOSKI, 2006), com mesmo nome do grupo.

O AMQP compreende tanto um protocolo de rede, que especifica o que clientes e servidores de mensagens devem enviar para interoperar entre si, quanto um modelo de protocolo, que especifica a semântica que uma implementação do AMQP deve obedecer para ser interoperável com outras implementações (VINOSKI, 2006). Os sistemas de mensagens assíncronas, normalmente, evocam imagens de sistemas de enfileiramento centralizado, monolítico, utilizando o algoritmo de escalonamento FIFO (*First in, First out* – primeiro a entrar, primeiro a sair), que aceitam mensagens em uma extremidade e

[11] Considerando funções implementadas em servidores externos SaaS, como os citados no texto.

as dispensam da outra. O AMQP adota uma abordagem mais modular, dividindo a tarefa de intermediação de mensagens entre trocas e filas de mensagens:

- Uma troca é essencialmente um roteador que aceita mensagens recebidas de aplicativos e, com base em um conjunto de regras ou critérios, decide para quais filas encaminhar as mensagens; trocas não armazenam mensagens, e
- Uma fila de mensagens armazena mensagens e as envia para os consumidores de mensagens. A durabilidade dos meios de armazenamento depende inteiramente das filas de mensagens de implementação da fila de mensagens – normalmente, armazenam mensagens no disco até que possam ser entregues, mas também são possíveis filas que armazenam mensagens puramente na memória.

Portanto, conforme descrito em Amazon Web Services, Inc. (2019), as filas de mensagens fornecem comunicação e coordenação para esses aplicativos distribuídos. As filas de mensagens podem simplificar significativamente a codificação de sistemas desacoplados, melhorando o desempenho, a confiabilidade e a escalabilidade destes. Ainda permitem que diferentes partes de um sistema se comuniquem e processem operações de forma assíncrona. Uma fila de mensagens mantém disponível um *buffer* leve que as armazena temporariamente, *endpoints* permitem que diferentes componentes de software estabeleçam conexão com a fila para o envio e o recebimento das mensagens.

Geralmente, as mensagens são pequenas e podem ser itens, como solicitações, respostas, mensagens de erro, ou apenas informações. Para enviar uma mensagem, um sistema chamado de produtor (*producer*) adiciona uma mensagem à fila (*queue*). A mensagem é armazenada na fila até que outro componente chamado de consumidor (*consumer*) recupere a mensagem e trate de acordo com sua lógica.

Figura 4 – Protocolo de fila de mensagens (MQ).
Fonte: Amazon Web Services, Inc. (2019).

Muitos *producers* e *consumers* podem usar uma mesma fila, mas cada mensagem é processada apenas uma vez, por um único *consumer*. Por esse motivo, normalmente, esse padrão de sistema de mensagens é chamado de comunicação direta ou ponto-a-ponto. Quando uma mensagem precisa ser processada por mais de um *consumer*, as filas de mensagens podem ser combinadas com um sistema de mensagens de publicação/assinatura em um padrão de projeto distribuído.

2.6 Trabalhos Relacionados

Há alguns estudos que cerceiam tópicos centrais deste trabalho, como em Balalaie, Heydarnoori e Jamshidi (2016), onde os autores relatam as experiências da migração incremental do *back-end* de um serviço *mobile* comercial, assim como a refatoração para uma arquitetura em microsserviços. Ao lado da migração e refatoração, o trabalho apresenta as práticas e como foram adotadas *DevOps*, as quais facilitaram um processo de migração suave para uma aplicação *cloud-native* através da arquitetura em microsserviços.

Em outro estudo, Chen (2018), o autor explicita a ânsia atual de responder as necessidades dos consumidores a velocidades sem precedentes. No trabalho, o autor expõe uma relação no desenvolvimento de software com o uso da arquitetura de microsserviços, entrega contínua e *DevOps* dos produtos. Mostrando, aplicado ao caso de uma companhia de apostas e jogos *online*, como o foco da entrega contínua e o *DevOps*, apoiado na arquitetura de microsserviços (que na época estava emergindo como uma opção de construção de software) dá-se como uma opção viável para o desenvolvimento de sistemas. Mostrando, também, quais são os pontos fortes e fracos, desafios e limitações. Concluindo de forma ponderada que a arquitetura pode introduzir desafios e complexidade, mas apresentando diversos pontos positivos que possibilitam potencializar o *DevOps* e a entrega contínua.

Quando levantadas produções sobre monitoramento, as produções acadêmicas normalmente focam no monitoramento do desempenho do software para habilitar práticas do *DevOps*, exemplificado em Waller, Ehmke e Hasselbring (2015). Neste caso, tem-se a inclusão de *frameworks* e *benchmarks* durante o processo de integração contínua do software. O autor sugere a inclusão dos *benchmarks* e monitoramento em etapas iniciais da produção para que seja possível adaptar a mudanças de forma ágil, aumentando, consequentemente, a efetividade da integração contínua do software e, subsequente, as práticas do *DevOps*.

Podemos encontrar de forma mais relacional ao trabalho quando em outros tipos de publicações, como é o caso da produção Palko (2015). Nesta produção, o autor apresenta áreas chave para monitoramento do *pipeline DevOps*, além de formatos para automatização, algumas práticas principais e exemplos de modelos. Nesse caso, incorpora-se aplicação do monitoramento, práticas para assegurar uma implementação mais completa do *DevOps* e, consequentemente, processos, produtos e infraestrutura mais estáveis, confiáveis e concisas.

Pode-se concluir que os demais trabalhos focam em aspectos específicos associados ao tema deste trabalho. Nesses casos, há um aprofundamento em seções ou partes específicas. Entretanto, cabe destacar que não foi possível encontrar na literatura pesquisa, contribuições no formato arquitetural com monitoramento no contexto do *DevOps*. Isto apresenta-se como oportunidade para a construção do projeto, assim como ressalta a im-

portância do mesmo, buscando unir dentre os trabalhos para formar uma pesquisa que agregue temas e seja base para trabalhos futuros dentro desta área.

3 Proposta

Conforme inicialmente levantado no capítulo introdutório (seção 1.1), este trabalho têm por objetivo construir um relato de experiências acerca das práticas *DevOps* (seção 2.1), além do monitoramento deste processo e dos sistemas, em arquiteturas orientadas a eventos, apoiando-se num desenho (*design*) arquitetural que faz uso de microsserviços (seção 2.2) e funções como serviço (seção 2.4.4).

O sistema a ser produzido bem como o tema proposto resumem-se a uma aplicação móvel para gerenciamento de pilotos e ligas de automobilismo virtual. Este produto proverá a demanda necessária pra a construção de sistemas, os quais serão objetos diretos do estudo proposto.

3.1 Problema

Consoante com a breve observação dos trabalhos relacionados (Seção 2.6), percebemos que há um interesse em averiguar o quanto o formato arquitetural interfere no ciclo de vida e na produção dos sistemas de software, em termos de *design* de microsserviços. Em um cenário onde o desenvolvimento de software está cada vez mais ágil, aberto a mudanças e customizável, há também uma pressão para a entrega de produtos com essas mesmas características em prazos cada vez mais curtos.

Práticas do *DevOps* buscam ajudar na agilização destes processos, trabalhando com *pipelines* de integração contínua de novas versões de software, com testes automatizados, com implantação e publicação também automáticas a cada novo incremento de produto. Entretanto, com o *design* de microsserviços, onde um produto de software é composto por uma série de outros produtos de software menores, ou então, por funções como serviços especialistas em processamentos específicos, surge incerteza se os processos serão mantidos idênticos, ou se as *pipelines* devem ser unidas.

Não se trata apenas de avaliar o processo de desenvolvimento do software e sua alteração a partir deste formato de *design*, mas sim todas as atividades adjacentes, para que então seja possível avaliar e construir produtos de software se atentando às necessidades do mercado.

3.2 Processo Metodológico

O processo metodológico aplicado no desenvolvimento do trabalho foi uma mescla de práticas ágeis. O uso do kanban deu-se pela sua flexibilidade, transparência e sua

aceitação dentro da implementação de metodologias ágeis no desenvolvimento de software (RADIGAN, 2015). Devido às características especiais do projeto, um kanban foi utilizado como uma ferramenta para facilitar o acompanhamento do desenvolvimento das atividades, na comunicação, na negociação rápida de mudanças e novos fatores inseridos no projetos, assim como no registro da completude do planejamento e proposta, trabalhando junto com as técnicas do processo ágil. O kanban foi introduzido (AHMAD; MARKKULA; OIVO, 2013) por volta de 1950 como um mecanismo de controle de fluxo para produção *just-in-time* (sob demanda).

Um quadro básico de kanban tem três passos presentes no fluxo, sendo o "*To Do*"(À fazer), "*Doing*"(Fazendo) e o "*Done*"(Feito) (RADIGAN, 2015). Entretanto, para este projeto, o quadro foi expandido para N passos no fluxo, sendo eles:

- *Project Backlog*: nesta coluna, estão presentes todas as tarefas já planejadas e construídas para o projeto mas que não foram priorizadas para a *sprint*;
- *Sprint Backlog*: nesta coluna, estão presentes todas as tarefas planejadas que devem ser executadas na *sprint* atual;
- *Doing*: neste fluxo, estão presentes as tarefas que estão sendo atualmente executadas.
- *QA (Quality Assurance)*: nessa faixa, têm-se as tarefas que já foram executadas e estão sendo validadas de acordo com os critérios de aceitação da tarefa (testes, completude, aceitação e etc.);
- *Done*: esta coluna arquivará todas as tarefas que já foram concluídas, entregues e aprovadas, e
- *Ice Box*: nesta coluna, estão as tarefas que apresentam algum impedimento ou foram planejadas para *sprint* atual e precisaram ser postergadas para outra.

Complementar ao uso do quadro kanban, o trabalho foi organizado e se deu por meio de práticas ágeis, iterativa e incremental. No mesmo processo, novos recursos e funcionalidades podem ser adicionados facilmente a partir do uso de múltiplas iterações (SHARMA; SARKAR; GUPTA, 2012). Importante frisar que devido à natureza da proposta do projeto, além de incrementos de software e sistemas, houveram tarefas de gerenciamento e infraestrutura, estes também são parte integrante do produto final, pois estão fortemente unidos com a proposta do projeto.

3.3 Escopo

O escopo de produção do trabalho atem-se às necessidades, expectativas e limitações conhecidas. Isto posto, as tarefas serão priorizadas a partir do que se espera como

sendo um resultado satisfatório para o cumprimento da proposta, dando prioridade aos produtos que trarão um retorno maior para alcançar o resultado dado como esperado.

Dividindo em áreas, podemos descrever o escopo a ser produzido em categorias, explanando o que se era esperado como produto a ser entregue ao final do estudo.

3.3.1 Infraestrutura

A infraestrutura desenvolvida no projeto foi o suporte para a execução dos sistemas e das tarefas de integração e entrega contínua do projeto. Consta também configurações dos sistemas e demais atividades relacionadas aos processos para funcionamento correto dos procedimentos relacionados às práticas do *DevOps* ou de natureza relacionada.

Logo, para o projeto, foi executado, em nível de infraestrutura, as tarefas apresentadas na Tabela 2.

Tabela 2 – Tarefas do escopo de infraestrutura.

Tarefas	
Infraestrutura	Configuração do servidor que hospedará os serviços
	Configuração do serviço de *DevSecOps* para cada serviços;
	Configuração do ambiente de entrega dos serviços;
	Criação do *pipeline* de *DevOps*;
	Criação dos repositórios dos serviços (e configuração adequada);
	Configuração do domínio do projeto.

Fonte: João Pedro Sconetto (2019)

3.3.2 Documentação

Para os artefatos e sistemas produzidos, buscou-se documentá-los devidamente, para que fosse possível prosseguir o trabalho realizado nos mesmos, assim como mantê-los.

Os documentos executados no escopo do projeto foram:

Tabela 3 – Tarefas do escopo de documentação.

Tarefas	
Documentação	Requisitos de serviços;
	Documentação dos serviços (configuração, dependências, testes, *API*, dentre outros);
	Documentação de configuração (infraestrutura);
	Documentação do *DevOps* (*pipeline* e configurações).
	Documento de arquitetura (representação e descrição);

Fonte: João Pedro Sconetto (2019)

3.3.3 Sistemas

Para a aplicação móvel, será construído um aplicativo prova de conceito. Uma aplicação que tem a integração com a arquitetura, mas apresenta somente as funções básicas da aplicação, sendo usada para validar os serviços e seu funcionamento. Em adição, este sistema foi dado com menor influência nos objetivos primários do estudo, o que favoreceu a decisão anteriormente acordada.

A *stack* de serviços da arquitetura, num desenho inicial, foi inicialmente prevista para ser composta de alguns serviços chave, sendo eles apresentados na Tabela 4.

Tabela 4 – Sistemas propostos para o escopo do projeto.

Sistemas	
Orquestrador	Um serviço responsável por receber requisições de execuções e por orquestrar o funcionamento dos demais serviços (funcionará com auxílio de fila de mensagens);
Autenticação	Um serviço responsável pelo registro e autenticação dos usuários dos serviços/aplicação;
Operador de Tempo	Um serviço responsável por operações que envolvem tempo para cálculos de tempo de voltas, diferença entre pilotos, entre outros;
Gerenciador de Campeonatos/Ligas	Um ou mais serviços para gerenciar, organizar e visualizar as ligas e campeonatos, além de outros dados e questões relacionados às mesmas[1];
Rastreador de Informações do Piloto	Um serviço para recuperar informações sobre o piloto a partir do perfil do mesmo no software simulador das corridas[2].

Fonte: João Pedro Sconetto (2019)

3.3.4 Outras Tarefas

Seguem outras tarefas, conforme colocado na Tabela 5, que foram levantadas para o pleno cumprimento da proposta.

Tabela 5 – Outras tarefas no escopo do projeto.

Tarefas	
Miscelânea	Definir e configurar o conjunto de informações e processos a serem monitorados;
	Definir e colher métricas das aplicações e práticas monitoradas;
	Descrever e relatar dados de descobertas do monitoramento e das execuções;
	Construir proposta com as práticas e orientações para o formato arquitetural deste estudo.

Fonte: João Pedro Sconetto (2019)

Posto isto, as limitações de projeto podem afetar a quantidade de métricas e de itens monitorados.

[1] Poderá haver mais de um sistema para esse gerenciamento devido ao fato de existir diversos itens a serem gerenciados, como pontuação, penalidades, pistas, carros permitidos e etc.

[2] As informações usadas serão específicas do simulador *Gran Turismo® Sport* a partir do perfil da *PlayStation® Network* dos usuários.

3.4 Arquitetura

A arquitetura da aplicação seguirá um desenho composto por diversos microsserviços especialistas em funções e cargas de trabalho. Quando possível e julgado necessário, microsserviços foram divididos e construídos como funções *FaaS*, sendo ainda mais focados em tarefas específicas de um dado processamento e/ou responsabilidade.

Na outra ponta da arquitetura, tem-se uma aplicação móvel, sendo esta tida como o *front-end* do produto e fará a interface entre o usuário final e os serviços que estarão sendo executados na *stack* de serviços. Seu desenvolvimento se deu com uso de tecnologias híbridas (como *React-Native*[3] e *Ionic*[4]) para implementação em ambas plataformas, Android e iOS.

Tem-se, a priori, de cinco a seis serviços, podendo estes serem divididos em outros serviços especialistas. Uma visão preliminar da arquitetura estaria disposta de acordo com a Figura ??.

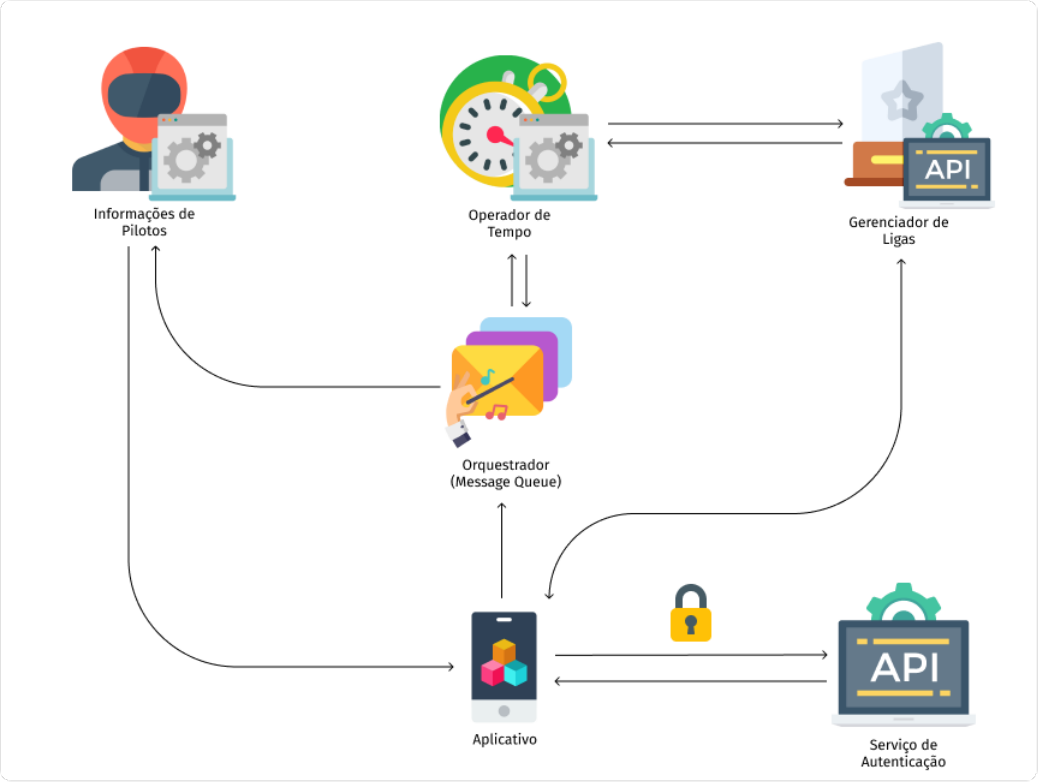


Figura 5 – Diagrama de contexto da arquitetura.
Fonte: João Pedro Sconetto (2019).

[3] https://facebook.github.io/react-native/ - Acesso em: 17 Fev. 2020
[4] https://ionicframework.com/ - Acesso em: 17 Fev. 2020

3.5 OpenFaaS

O OpenFaaS (ELLIS, 2017b; ELLIS, 2017a) é uma plataforma (*framework*) *open source* para a construção, implantação e uso de funções *serverless* (*FaaS*) com uso de contêineres *Docker* ou *Kubernetes*. O princípio desta ferramenta é que – *"qualquer equipe pode, facilmente, ter em sua infraestrutura o stack necessário para construir e executar funções como serviço"*. O OpenFaas fornece uma arquitetura estruturada e sem necessidade de conhecimento específico para a instalação deste formato de arquitetura, podendo dividir o *stack* em três camadas (*bottom-up)*:

- Camada de Infraestrutura: Nesta camada, têm-se as tecnologias e os serviços que dão base estrutural para que o OpenFaaS seja capaz de operar as funções;
- Camada de Aplicação: Nesta camada, temos aplicações que fazem o controle, monitoramento, direcionamento, ou seja, aqui estão alguns serviços que darão suporte para o correto funcionamento da arquitetura, e
- Camada GitOps/IaaC: Nesta camada, estão os serviços que trabalham em conjunto com o OpenFaaS para que as funções sejam *Cloud Native*, isto é, as operações integração e entrega sejam via GitOps, totalmente automáticas por ferramentas presentes em provedores e configuravéis por processos dentro do servidores de ferramentas de controle de versionamento.

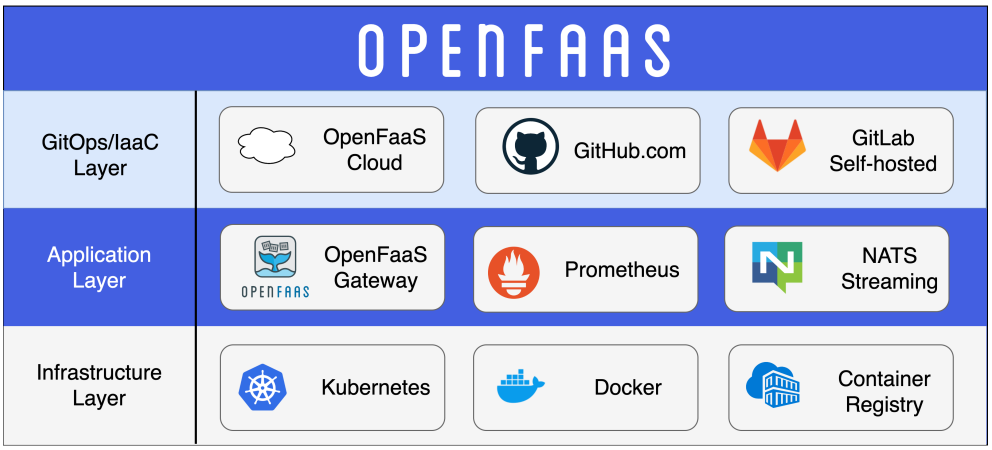


Figura 6 – *Stack* da estrutura do OpenFaaS
Fonte: Ellis (2019).

O OpenFaaS opera com um conjunto de ferramentas, como visto na *stack* ilustrada na Figura ??. Desta forma, as ferramentas e os serviços cooperam entre si para a execução dos serviços solicitados. Na Figura ??, tem-se exemplificado um fluxo de trabalho do OpenFaaS quando uma função é solicitada/executada.

Na estrutura da arquitetura, temos (ELLIS, 2019) o *gateway*, que pode ser acessado através de sua API REST, via CLI ou através da interface do usuário (ao executar a *stack*

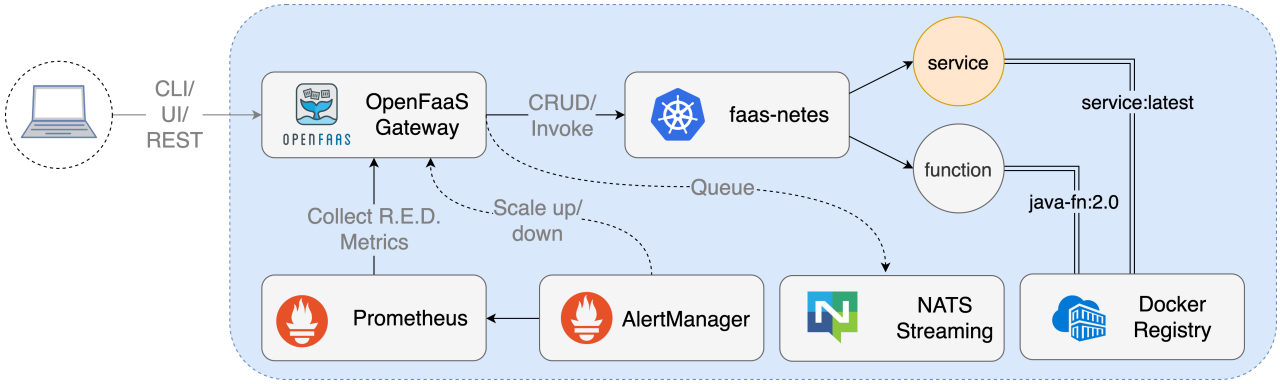


Figura 7 – Fluxo de trabalho para tratamento de uma requisição pelo o *OpenFaaS*
Fonte: Ellis (2019).

do OpenFaaS, ele provê em uma URL, uma interface de usuário numa página web para o uso das funções). Todos os serviços ou funções são expostos a uma rota padrão, mas os domínios personalizados também podem ser usados para cada nó de extremidade.

O **Prometheus** coleta métricas disponíveis por meio da API do *gateway* e usadas para dimensionamento automático (escalando e replicando automaticamente, *scale up* ou *scale down*).

Alterando a URL de uma função de `/function/NAME` para `/async-function/NAME`, uma chamada pode ser executada em uma fila usando o **NATS Streaming**. Você também pode transmitir uma URL de retorno de chamada opcional para o uso *callbacks* (chamada de retorno) de chamadas assíncronas.

O **faas-netes** é o provedor de orquestração mais popular do OpenFaaS, mas o Docker Swarm, o Hashicorp Nomad, o AWS Fargate/ECS e o AWS Lambda também estão disponíveis. Dessa forma, a partir do provedor escolhido e configurado será o orquestrador utilizado para buscar a imagem e o serviço para a execução da função.

3.6 Docker

O Docker é uma plataforma (Docker Inc., 2019) para desenvolvedores e *sysadmins* construirem, compartilharem e rodarem aplicações por meio de contêineres. O contêiner pode ser definido (MERKEL, 2014) como o equivalente de uma máquina virtual leve, ou seja, um contêiner roda nativamente em ambientes Linux e compartilha o kernel do ambiente hospedeiro (*host*) com os outros contêineres. Ele é executado como um processo discreto, não tomando mais memória do que um outro executável qualquer, tornando-o leve.

Fundalmentalmente, (MERKEL, 2014) um contêiner docker utiliza da tecnologia LXC (*LinuX Container*). O LXC usa *namespaces* em nível de kernel para isolar os contêi-

neres uns dos outros e do *host*. O *namespace* do usuário separa o banco de dados do *host* e do contêiner, garantindo, desta forma, que o usuário *root* (administrador) do contêiner não tenha privilégios de administrado na máquina hospedeira. O *namespace* do processo é responsável por mostrar e gerenciar apenas os processos em execução no contêiner, não no *host*. Por fim, o *namespace* da rede provê ao contêiner seu próprio dispositivo de rede e endereço IP virtual.

Um dos aspectos importantes (Docker Inc., 2019) do isolamento do contêiner é que cada contêiner interage com seu próprio, e privado, sistema de arquivos. Este sistema de arquivos é provido por uma imagem Docker. Uma imagem inclui tudo que é necessário para rodar a aplicação – o código ou o binário, dependências de execução e qualquer outro objeto do sistema de arquivos necessário. Podemos ver na Figura 8 uma comparação estrutural em como é disposto um contêiner e como é disposto uma máquina virtual.

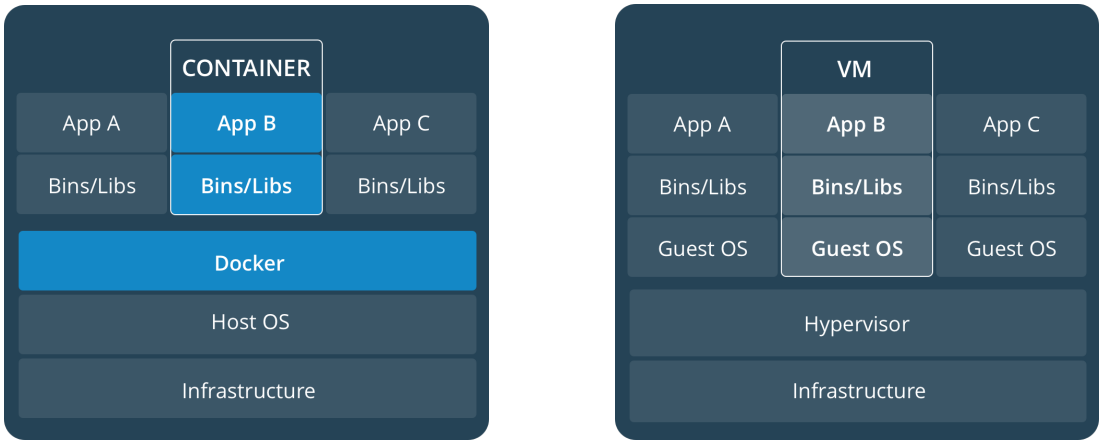
Com este formato de execução, o Docker auxilia e resolve alguns problemas presentes no processo de construção e implantação de sistemas de software. Conforme colocado pelo autor em (MERKEL, 2014), podem ser citados três deles:

- Conflito de dependências: um serviço usa o banco PostgreSQL 9 e outro o PostreSQL 10, não há problema, execute contêineres separados com as versões necessárias;
- Ausência de dependências: cada serviço necessita de uma série de dependências e a cada nova instanciação é preciso instalá-las. Com o Docker, as dependências já vem atreladas ao contêiner, e
- Diferença de plataformas: trocar de distribuições, e até sistemas operacionais, não são mais um problema. Caso a plataforma execute Docker, os contêineres serão executados sem problema algum.

Em contraste e comparação, uma máquina virtual (VM) (Docker Inc., 2019) executa um sistema operacional "convidado"com toda a sua estrutura, fazendo o acesso virtual aos recursos do hospedeiro por meio de um *hypervisor*. Em geral, as VMs incorrem em muita sobrecarga além do que está sendo consumido pela lógica do seu aplicativo.

3.7 Kubernetes

O Kubernetes é uma plataforma (The Linux Foundation, 2019) portátil, extensível e *open source* para gerenciar cargas de trabalho e serviços em contêineres, que facilita ambos automação e configuração declarativa. O Kubernetes surgiu do projeto de mesmo nome criado pelo Google em 2014 (BURNS et al., 2016), que teve seu código-fonte aberto para a comunidade. O Kubernetes foi construído com base em uma década e meia de



(a) Estrutura Docker (b) Estrutura de Máquina Virtual

Figura 8 – Diferença estrutural entre Docker e Máquinas Virtuais

Fonte: Docker Inc. (2019)

experiência que o Google tem na execução de cargas de trabalho em ambientes de produção em grande escala, combinadas com ideias e práticas da comunidade.

O Kubernetes apoia-se na tecnologia de contêineres (Seção 3.6) para o seu funcionamento. Os contêineres são uma boa maneira de agrupar e executar aplicações (The Linux Foundation, 2019). Em um ambiente de produção, onde é necessário gerenciar os contêineres que executam as aplicações e garantir que não haja tempo de inatividade, o Kubernetes encaixa-se como ferramenta. Por exemplo, se um contêiner for interrompido, outro contêiner precisa ser iniciado. Esse formato é como o Kubernetes trabalha, tratando esses eventos em nível de sistema, fornecendo uma estrutura para executar sistemas distribuídos de forma resiliente. Ele cuida do dimensionamento e do *failover*[1] das aplicações, fornece padrões de implantação e muito mais. O Kubernetes pode, por exemplo, gerenciar facilmente uma implantação de canário para o sistema.

O Kubernetes têm em seu núcleo (BURNS et al., 2016) um armazenamento persistente compartilhado, com componentes observando alterações em objetos relevantes, em casos mais comuns os cointêineres. Os estados são acessados exclusivamente por meio de uma API REST de um domínio específico que aplica versionamento, validação, semânticas e políticas em um nível abstrato mais alto, em suporte a uma matriz mais diversa de clientes. Ressaltando que o Kubernetes foi desenvolvido fortemente focado na experiência de desenvolvedores que construíam aplicações que executavam em *clusters*[2]. O objetivo principal do *design* era deixar fácil a implantação e o gerenciamento de sistemas distribuí-

[1] *Failover* é uma técnica de tolerância a falhas onde um servidor, sistema ou serviço é comutado para um destes componentes redundante em caso de falha ou término de execução anormal daquele previamente ativo.

[2] Um *cluster* consiste em computadores fracamente ou fortemente ligados que trabalham em conjunto, de modo que, em muitos aspectos, podem ser considerados como um único sistema.

Tabela 6 – Serviços e funcionalidades providas pelo Kubernetes.

Descoberta de serviços e balanceamento de cargas	O Kubernetes pode expor um contêiner usando o DNS ou usando seu próprio endereço IP. Se o tráfego para um contêiner for alto, o Kubernetes poderá equilibrar a carga e distribuir o tráfego da rede para que a implantação seja estável.
Orquestração de armazenamento	O Kubernetes permite montar automaticamente um sistema de armazenamento de sua escolha, como armazenamentos locais, provedores de nuvem pública, entre outros .
***Rollout* e *rollbacks* automatizados**	Você pode descrever o estado desejado para seus contêineres implantados usando o Kubernetes, e ele pode alterar o estado real para o estado desejado a uma taxa controlada. Por exemplo, você pode automatizar o Kubernetes para criar novos contêineres para sua implantação, remover os contêineres existentes e adotar todos os seus recursos para o novo contêiner.
Gerenciamento automático das aplicações	Você fornece ao Kubernetes um *cluster* de nós que ele pode usar para executar tarefas em contêiner. Você diz ao Kubernetes quanta CPU e memória (RAM) cada contêiner precisa. O Kubernetes pode ajustar contêineres nos seus nós para fazer o melhor uso de seus recursos.
***Self-healing* (tolerância a falhas)**	O Kubernetes reinicia os contêineres que falham, substitui os contêineres, mata os contêineres que não respondem à verificação de integridade definida pelo usuário e não os disponibiliza aos clientes até que estejam prontos para servir.
Segredos e gerenciamento de configuração	O Kubernetes permite armazenar e gerenciar informações confidenciais, como senhas, tokens *OAuth* e chaves SSH. Você pode implantar e atualizar segredos e configuração de aplicativos sem reconstruir suas imagens de contêiner e sem expor segredos na configuração da *stack*.

Fonte: The Linux Foundation (2019)

dos complexos, enquanto se benificiar da utilização melhorada que a compartimentação (contêinerização) permite.

Algumas funcionalidades, serviços e facilidades que o Kubernetes provê de experiência imediata pode ser listado conforme descrito na Tabela 6.

3.8 Outras Ferramentas

Algumas outras ferramentas foram selecionadas para uso no projeto, sendo estas descritas na Tabela 7, pois não julgou necessário explicação técnica por serem, ou amplamente definidas, ou amplamente conhecidas, ou pelo seu uso não ser protaganista dentro na proposta, ainda que necessária.

Tabela 7 – Ferramentas utilizadas no projeto.

Ferramenta		
Nome	Natureza	Uso
JavaScript	Linguagem de Programação	Construção de sistemas.
Pyhton	Linguagem de Programação	Construção de sistemas.
Git	Controle de Versionamento	Versionamento de sistemas, documentos, configurações e demais artefatos.
GitHub	Repositório de Código	Hospedagem do código, gerenciamento das atividades (*issues*), publicação de pacotes e etc.
Roadmunk	Gerência de Projeto	Registro do *roadmap* de atividades.
Whimsical	Documentação de Projeto	Registro de diagramas e processos de execução do projeto.
Figma	Documentação de Projeto	Registro e diagramação da arquitetura do projeto.
Telegram	Comunicação	Comunicação entre orientador/autor.
DigitalOcean	Infraestrutura	Servidor para implantação e execução dos sistemas.
DataDog	Monitoramento	Serviço de monitoramento de aplicações em nuvem por meio de plataforma de análise de dados baseadas em SaaS.
Travis CI	Desenvolvimento/Integração	Serviço de integração e entrega contínua de software.
Codecov	Desenvolvimento/Integração	Serviço de análise e relato de código.
DockerHub	Desenvolvimento/Entrega	Serviço de publicação de imagens/contêineres *docker*.

Fonte: João Pedro Sconetto (2020)

Durante a execução do projeto houve algumas trocas, inserções e remoção de ferramentas, quando comparado este conjunto descrito com o conjunto apresentado durante a primeira parte deste trabalho. As modificações feitas foram informadas e acordadas, e o intuito das mesmas eram acelerar e/ou melhorar o resultado durante a execução do projeto.

4 O projeto

4.1 Implementação

Nesta seção é apresentado os resultados e relatos da execução das tarefas do projeto, cada um dentro do escopo específico do levantamento inicial (exposto na proposta). Buscou-se exibir em ordem na qual as informações pudessem ser lidas linearmente.

4.1.1 Requisitos

Para o levantamento de requisitos no projeto foi realizado, ao final de 2019, uma copa de automobilismo virtual com a finalidade de simular as dificuldades que um gerente/gestor deste tipo de evento enfrenta. A documentação produzida para a copa pode ser vista nos anexos B, e C. Nestes documentos estão descritas as informações dos eventos da copa, assim como as informações e tabela de resultado destes.

O autor, como gestor da liga, ficou responsável por organizar todas as corridas, executá-las, registrar todos resultados e, se necessário, registrar problemas. A Tabela 8 apresenta os épicos relacionados. Sendo os identificadores iniciados com "E" indicando épicos para funcionalidades de sistema/usuário, e os iniciados com "T" indicando épicos técnicos.

Tabela 8 – Épicos do *Project SRC*.

Identificador	**Épico**
E1	Gerenciar usuários do sistema (pilotos, gestores, administradores e comissários)
E2	Gerenciar ligas (participantes, pontuações, pistas e relacionados)
E3	Gerenciar times e contratações
E4	Operador de tempo (auxiliar a cronometração de corridas)
E5	Aplicação de interface para usuário dos serviços (aplicação móvel híbrida)
E6	Construir integração com software de simulação de automobilismo (Gran Turismo™ Sport)
T1	Definir identidade visual do projeto
T2	Construir a *pipeline* de *DevOps* (CI/CD) do projeto
T3	Definir documentação básica do projeto
T4	Construir e provisionar infraestrutura para execução do projeto

Fonte: João Pedro Sconetto (2019)

4.1.2 Modelo de Dados

Com a finalidade de facilitar o entendimento do domínio do projeto e favorecer a construção dos sistemas, principalmente a API, que na versão atual da arquitetura é o microsserviço de maior importância, foi confeccionado um modelo de entidade-relacionamento com as classes de objetos do projeto.

O modelo de dados representa as classes necessárias para o desenvolvimento dos épicos "E1", "E2" e "E3". Os atributos definidos para cada entidade estão no Anexo D. É oportuno citar que para o desenvolvimento do épico "E6", uma evolução do modelo será necessária.

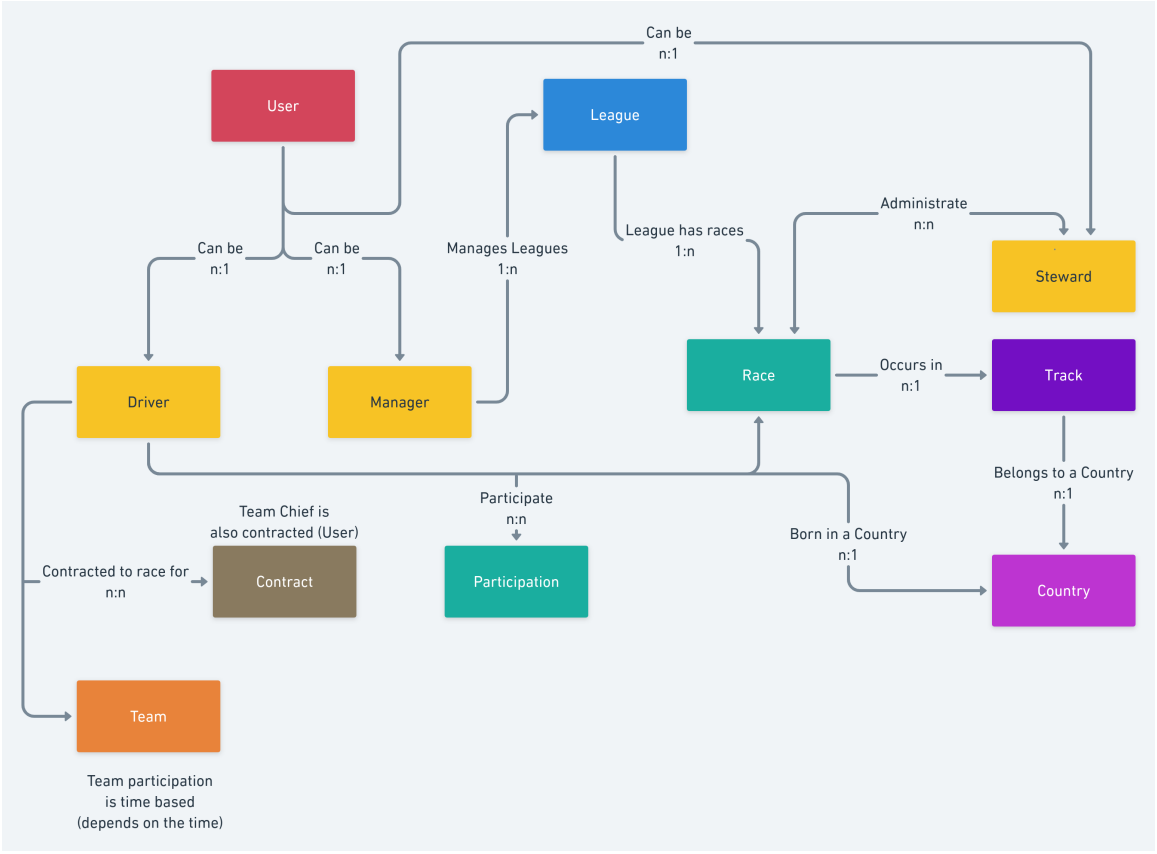


Figura 9 – Modelo geral de dados (recurso/classe) do *Project SRC*.
Fonte: João Pedro Sconetto (2020).

As principais entidades são vistas em cores distintas, como é o caso da entidade *User* (Usuário) em vermelho e suas sub-entidades, que são as especializações de usuário, em amarelo. A mesma lógica vale para as demais entidades.

4.1.3 *DevOps*

O processo de *DevOps* deu-se no formato do *pipeline* descrito na Figura 10. Para descrever as fases que ocorrem durante a execução do *pipeline*, decidiu-se por dividir as fases em estágios, buscando separar logicamente e cronologicamente eventos que dão-se

no processo. Estágios tem começo e fim definidos, assim como produtos/recursos gerados ao final de cada um. Caso ocorra, e se necessário for, estágios podem ser executados n vezes, ou retornar a estágios anteriores.

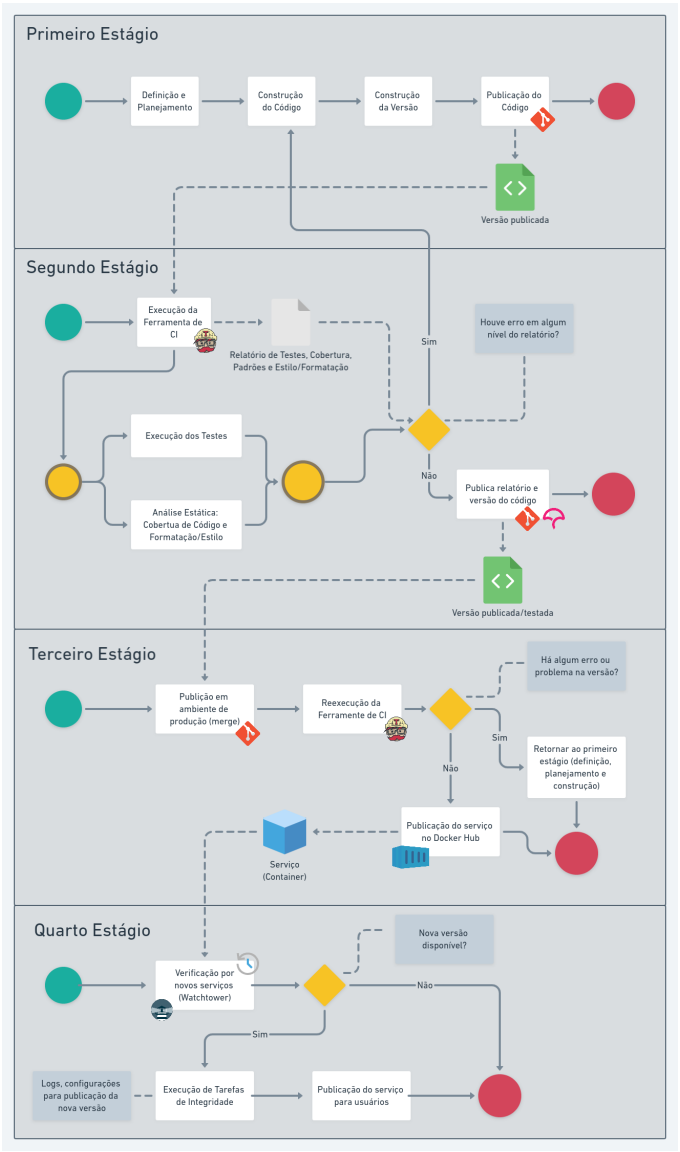
Há, portanto, quatro estágios bem definidos, e verificam-se como descrito a seguir.

O primeiro estágio, trata-se da fase inicial do desenvolvimento da nova *feature* ou correção. Uma atividade de definição e planejamento desenha o escopo da execução. Conseguinte dá-se a construção do código para adição definida. Cria-se uma nova versão (*commit(s)* de uma proposta de adição), e faz-se a publicação desta no repositório relacionado. Neste estágio, é inusual ser necessário execuções repetidas sem que se prossiga para o próximo estágio, e o produto de saída é uma versão publicada do produto de software.

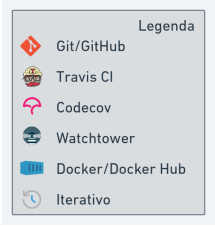
No segundo estágio, temos a primeira execução automatizada. A publicação da versão desencadeia a execução da ferramenta de integração contínua (CI - *Continuos Integration*), onde serão efetuados testes e análises (com foco em análise estática e de estilo de código/padrões de codificação). Tem-se o primeiro produto intermediário, o relatório da ferramenta de integração que será examinado e enviado para a atividade conseguinte, para o Codecov (ferramenta de análise descrita na Tabela 7). Em ocorrência da identificação de algum erro, sejam nos testes ou na análise da versão publicada, retorna-se ao estágio anterior. O produto de saída é a mesma versão do produto, analisada e testada (índices dos testes e análises são automaticamente atualizados e representados na documentação do sistema por meio de um *read me*).

O terceiro estágio, toma-se a versão publicada do estágio anterior, faz a mescla (*merge*) do código em ramificações (*branches*) do repositório no ambiente de homologação e/ou produção. Mais uma vez a ferramenta de integração contínua é acionada, operando tarefas para a construção e entrega de versões estáveis do software/serviço, no formato de um contêiner *docker*. Caso ocorra qualquer problema durante a execução, as versões não serão construídas, e faz-se necessário uma análise manual e retorno ao primeiro estágio. O produto de saída é o citado acima, contêiner com a versão estável do software. Esse será publicado em um repositório de contêineres, o DockerHub.

Por fim, o quarto estágio, que ocorre nos servidores do projeto, cuida da fase final da entrega da nova versão do software. O serviço *Watchtower* fica responsável por fazer verificações periódicas no repositório DockerHub por novas versões do software. Caso exista uma nova versão disponível, o *Watchtower* efetiva a transferência da mesma, executa tarefas administrativas, como publicação em *log* dos eventos e configurações inerentes à execução dos contêineres, substituindo a versão antiga pela nova. Deste modo, encerra-se o último estágio e, consequentemente, o *pipeline* do projeto.



(a) Estrutura Docker



(b) Legenda

Figura 10 – Processo do *pipeline* de *DevOps* da aplicação

Fonte: João Pedro Sconetto (2020).

4.1.4 Arquitetura

Durante o desenvolvimento do projeto, a arquitetura foi alterada para um formato mais simples, mas ainda seguindo o padrão de microsserviços e *FaaS* orientados a eventos. Podemos perceber na Figura ??, representando a versão atual da arquitetura, quando comparada com a arquitetura planejada, da Figura ??. Os serviços do formato API *Rest* foram unificados, decidiu-se por não desenvolver nesta fase o sistema de informação de pilotos (integração com o sistema de informações do Gran Turismo™ Sport), assim como o orquestrador de mensagens. Há a adição de um novo microsserviço, o *Rethink Database Manager*, que é utilizado para gerenciar a comunicação dos demais serviços com o banco de dados do projeto.

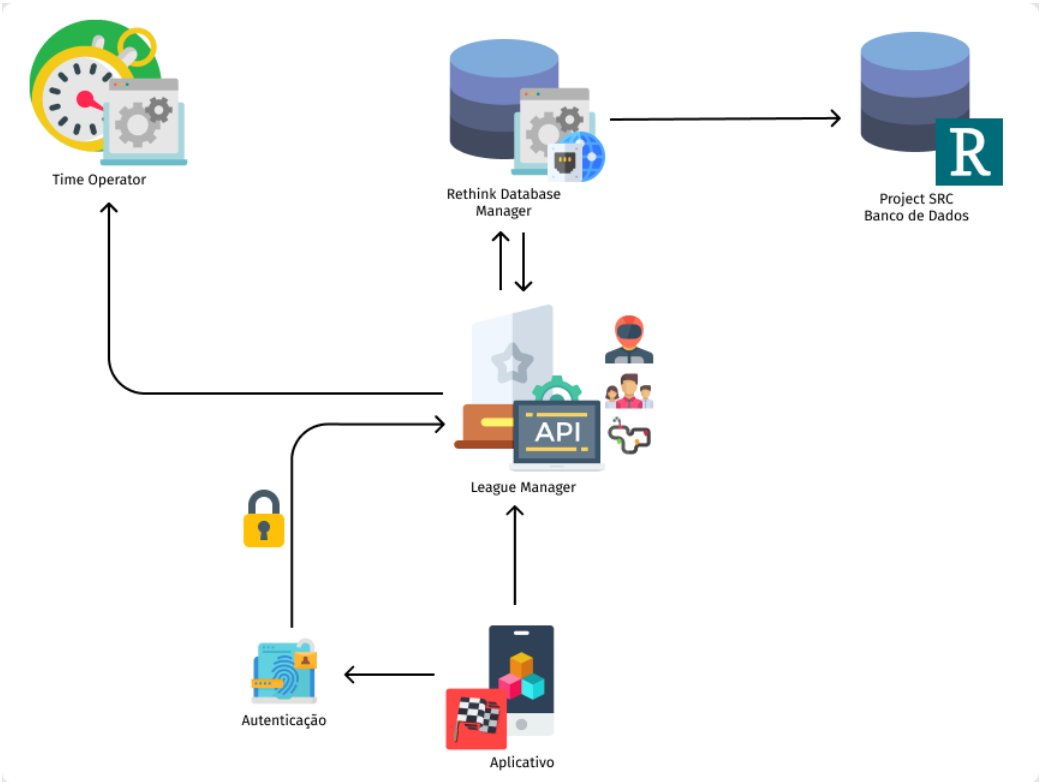


Figura 11 – Diagrama de contexto do *Project SRC*.
Fonte: João Pedro Sconetto (2020).

A construção do sistema de informações dos pilotos e do orquestrador foram postergadas para execuções futuras do projeto, e crê-se que a integração destes novos sistemas com a arquitetura atual não será obstáculo, dada a natureza de funcionamento dos sistemas desenvolvidos, assim como a própria arquitetura.

Logo, os sistemas/serviços desenvolvidos, com uma descrição da sua finalidade e formato de funcionamento, foram:

- *Time Operator*: Um função como serviço para realizar operações matemáticas sobre tempo, como estimar o tempo de corrida a partir do número de voltas pelo o tempo

médio de pista. Utiliza do *framework* do *OpenFaaS* e foi construído em Python;

- *Rethink Database Manager*: Um microsserviço para gerenciar operações de banco de dados assim como comunicação deste com outros serviços. Utiliza da tecnologia de *WebSocket* onde recebe e envia dados, e foi construído em Python;
- *League Manager*: Um microsserviço para gerenciar os demais serviços que funciona com *backend* da arquitetura (gerenciando e utilizando os demais serviços quando necessário), além de embarcar as lógicas de negócio, e de dados do projeto. Utiliza do *framework* FastAPI e construído Python, e
- *Project SRC App*: Aplicação híbrida para Android e iOS que serve como interface para o usuário com os serviços disponibilizados do projeto. Construído em JavaScript com o uso do *framework* React-Native. Contém um *backend* próprio, que é reduzido quando comparado com os demais sistemas, que lhe permite conversar com a API do *League Manager*.

A única dependência externa da arquitetura é o banco de dados *RethinkDB*. Trata-se de um banco de dados orientado a documentos (no qual, faz-se uso de documentos JSON), escalável por configuração e focado em aplicações *real-time*. Uso destas funcionalidades do BD com o *WebSocket* do serviço de gerenciamento de dados mitigam problemas de uso intenso da aplicação.

Está presente na aplicação apenas funcionalidades básicas que corroboram o mesmo como prova e que apresentam a viabilidade da arquitetura em geral. Portanto, o funcionamento das funções básicas do app, em certo nível, implica no funcionamento da arquitetura, evidênciando a comunicação entre os sistemas.

Nas Figuras ??, ?? e ??, têm-se as capturas de telas da versão final do protótipo da aplicação. As telas básicas são as telas de registro, de autenticação e um primeiro *design* da tela inicial do usuário, com algumas informações de um perfil de piloto.

Um pequeno vídeo de apresentação da aplicação pode ser encontrado hospedado no serviço de vídeos *Youtube* na seguinte URL: <https://bit.ly/3k2JFIL>. Uma outra URL relevante é a organização do projeto onde se encontra o código fonte dos sistemas, que pode ser acessada na URL: <https://github.com/Project-SRC/>.

4.1.5 Monitoramento

Para fechar o ciclo da proposta do projeto, foi implementado o monitoramento e agregação de *logs* dos sistemas/serviços com o auxílio da ferramenta *Datadog*, conforme apresentado na Tabela 7, das ferramentas utilizadas no projeto. O monitoramento deu-se para avaliar eventos da integração contínua, além de apresentar informações sobre o funcionamento da arquitetura. Outrossim, a implementação do monitoramento serviu também

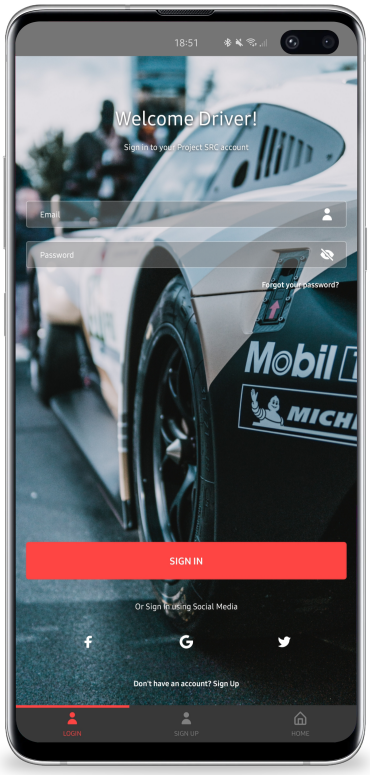


Figura 12 – Protótipo de alta fidelidade da tela de registro do aplicativo *Project SRC*.
Fonte: João Pedro Sconetto (2020).

como um formato de comprovação de funcionamento da arquitetura, sendo possível observar eventos de comunicação entre os sistemas.

O painel de monitoramento de serviços e infraestrutura apresenta informações e *logs* provindos do *socket docker* de funcionamento. O painel dispõe de informações de CPU, memória, rede e uso de dados (leitura e escrita no *hardware*). Suplementando os dados da infraestrutura, o painel expõe eventos que ocorrem no contêineres, serviços ativos e uso de cada serviço pelo tempo de CPU de cada contêiner. A Figura ?? exibe uma captura de tela do painel, onde todos os serviços estão implantados e executados.

Isolado do painel de monitoramento de serviços e infraestrutura, há um painel para o banco de dados, este que foi utilizado para verificar o uso e desempenho do banco de dados. Neste painel, têm-se informações como clientes conectados ao banco de dados, leituras por segundo, escrita por segundo (os dois últimos separados por tabela, servidor e estado da réplica), quantidade de servidores executando (em caso de múltiplos) e algumas outras informações de sanidade do serviço. Na Figura ??, podemos ver o painel de monitoramento citado, em um ocioso do banco de dados.

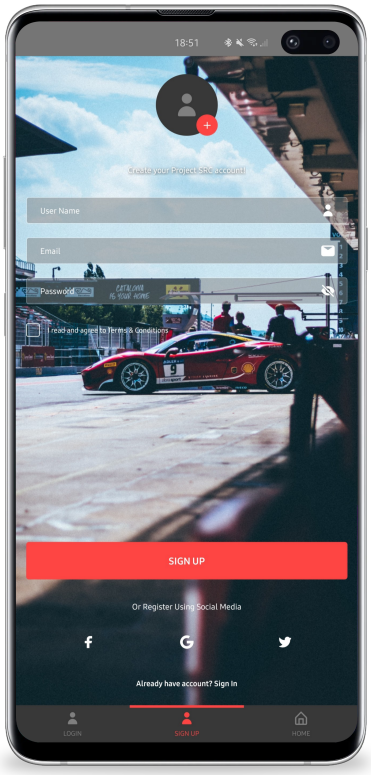


Figura 13 – Protótipo de alta fidelidade da tela *login* do aplicativo *Project SRC*.
Fonte: João Pedro Sconetto (2020).

4.2 Lições Aprendidas

Através da execução do estudo com a construção do *Project SRC*, alguns padrões emergiram nos domínios das tarefas desenvolvidas e, com a entrega da versão atual da solução, foi possível averiguar e absorver diversos conhecimentos, os quais estão relatados a seguir. Ressaltando que, o contexto específico, no qual o projeto esteve inserido, foi o da construção de uma arquitetura de microsserviços, apoiada na cultura do *DevOps* (implementado entrega e integração contínua), com atividades de segurança e monitoramento. Portanto, trabalhos futuros ou até trabalhos relacionados, podem se beneficiar dos aprendizados desta execução.

Um primeiro ponto beneficial é o fato de que durante o desenvolvimento é possível construir uma base de conhecimento, seja da arquitetura, dos processos, dos *frameworks* e das ferramentas, o que permite e facilita a construção da solução. Entretanto, acredita-se que, em um nível básico e com apropriado gerenciamento de conhecimento, este benefício pode ser usufruído em diversos tipos de projetos. Dessa forma, uma boa prática, em geral, é internalizar em projetos uma estratégia de gerenciamento e propagação de conhecimento.

Entre alguns benefícios próprios do contexto do projeto, pode-se mencionar que, ao trabalhar com uma arquitetura de microsserviço, usualmente, as responsabilidades de

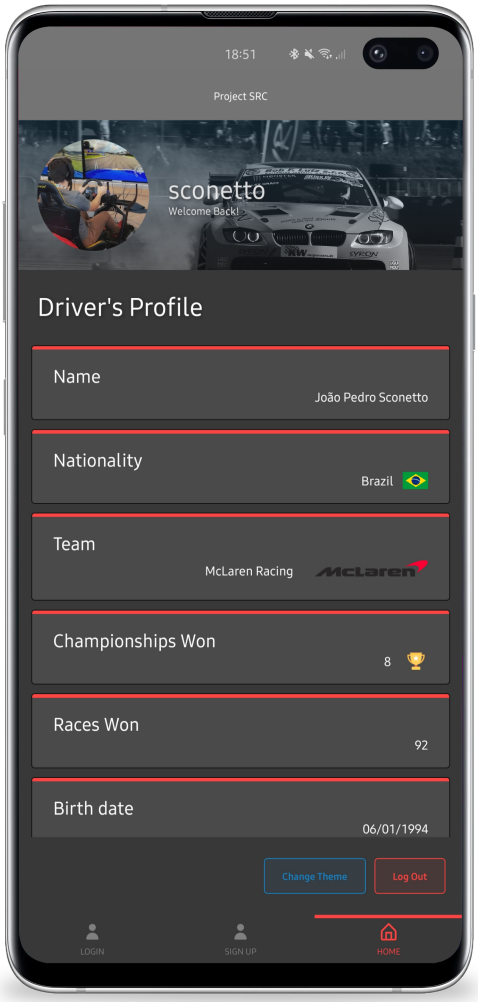


Figura 14 – Protótipo de alta fidelidade da tela inicial do aplicativo *Project SRC*.
Fonte: João Pedro Sconetto (2020).

cada sistema são em menor quantidade, isto é, os serviços são especialistas em poucas ou apenas uma atividade (funções como serviço), o que os torna menores, demandam menos tempo para produção e facilmente replicáveis. Consequentemente, atividades como teste, configuração, implantação, entre outras, tornam-se descomplicadas, podendo até serem, em parte ou em todo, replicadas em sistemas similares. Logo, todo recurso produzido têm potencial de transformar-se em fragmento reciclável e reusável no projeto. Isso vale, também, para as atividades operacionais, onde muitas vezes um fragmento, como uma configuração de um contêiner *docker*, precisa ser construído uma vez, contudo, pode ser replicado em vários serviços que são um contêiner.

Com o ganho na agilidade de construção e desenvolvimento, marcos e entregas do ciclo de vida do projeto podem ser alcançadas mais rapidamente, quando comparadas com outros formatos que demandam o desenvolvimento de sistemas com múltiplas responsabilidades. Salientando e reforçando que, a definição das divisões de responsabilidade,

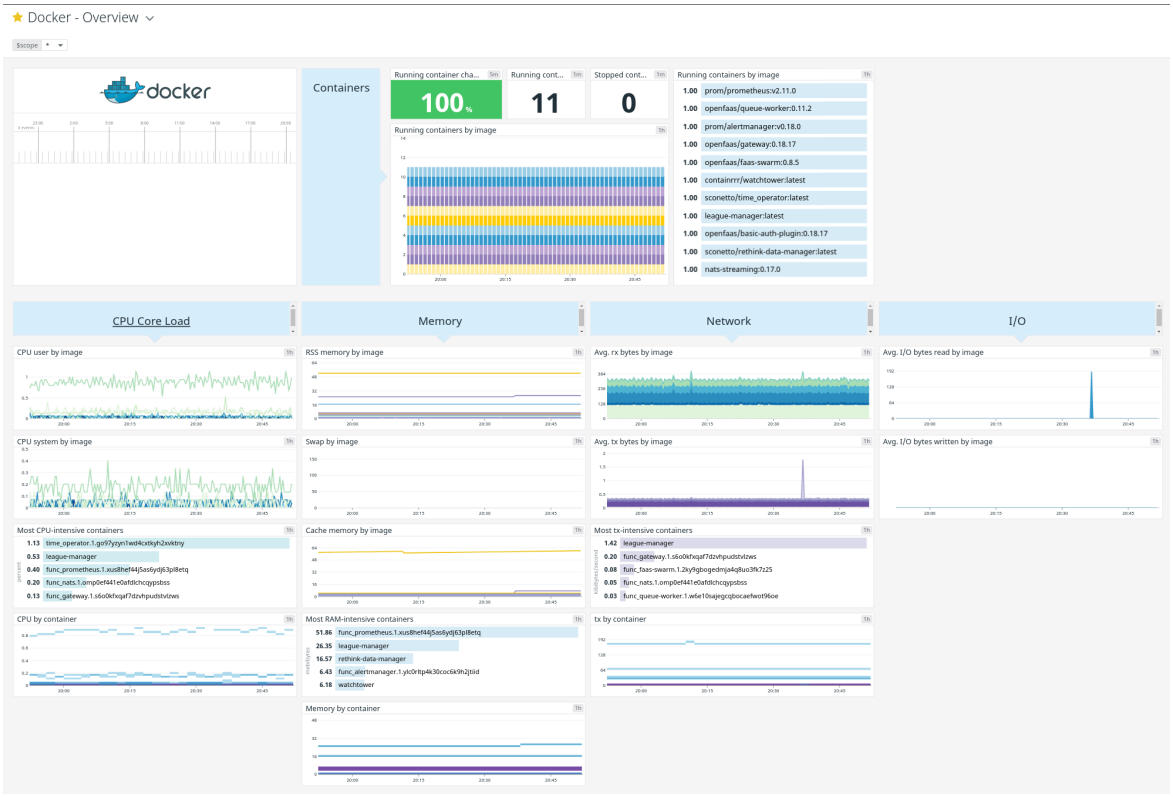


Figura 15 – *Dashboard* de monitoramento da infraestrutura e de serviços do projeto.
Fonte: João Pedro Sconetto (2020).

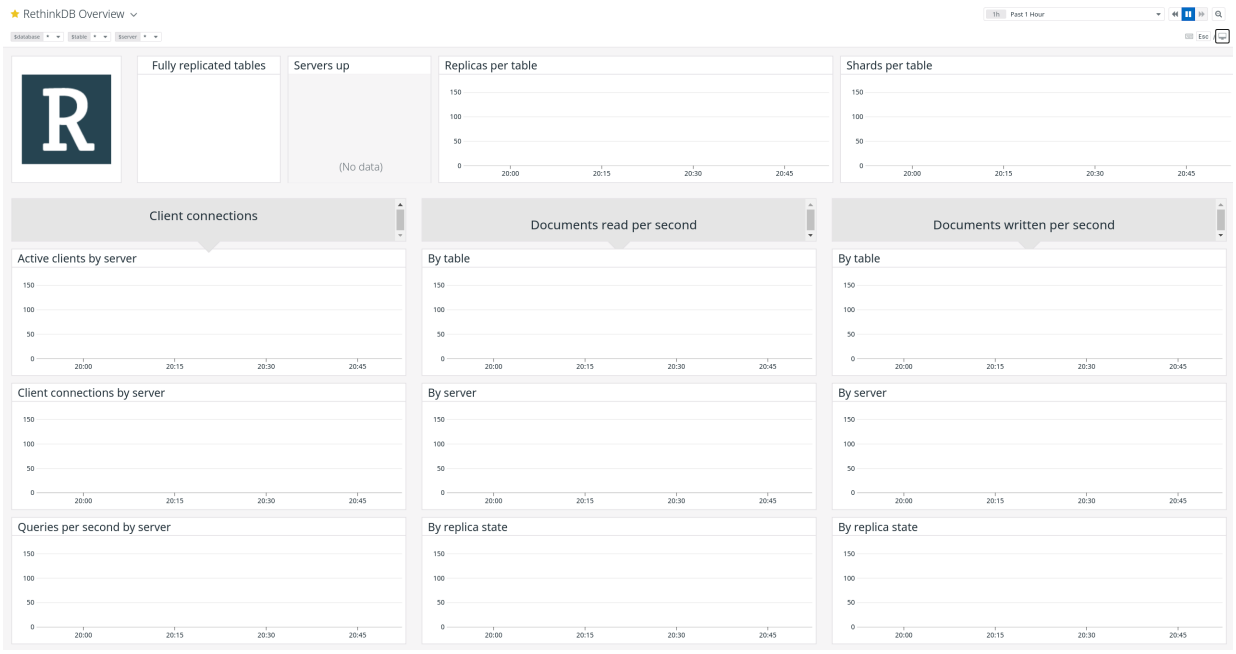


Figura 16 – *Dashboard* de monitoramento de uso e desempenho do BD *Rethink*.
Fonte: João Pedro Sconetto (2020).

para o desenho arquitetural, é tão importante quanto a construção do software, sendo esta uma boa prática. Porém, a habilidade de definir precisamente as divisões vêm com o gerenciamento do conhecimento e com experiência dos que participam do projeto.

O monitoramento aciona e ajuda a manter a agilidade no desenvolvimento, provendo informações que habilitam uma tomada de decisão consciente, podendo rapidamente resolver dificuldades ou mesmo servir como suporte para evoluções futuras. Auxiliando, similarmente, a identificar problemas e tratá-los velozmente.

Contudo, o formato também apresenta alguns empecilhos que devem ser levados em consideração com seu uso. A cada novo serviço com uso de uma nova ferramenta, *framework* ou linguagem, pode requerer a construção de uma nova base de conhecimento no projeto, tornando o desenvolvimento do sistema em questão lento. Outra adversidade, e que muitas vezes é a razão da falha na implementação destas arquiteturas, é a falta de cuidado no planejamento e definição da comunicação dos serviços, em outras palavras, é necessário estabelecer estratégias para a comunicação de toda arquitetura, para garantir que os serviços possam funcionar graciosamente entre si e atingir o sua finalidade, pois não existe razão que justifique a construção do formato arquitetural se o software não soluciona o problema que se propõe a solver.

5 Conclusão

Com o crescimento da indústria de *hardware*, trazendo a mãos de usuários dispositivos com capacidades computacionais, a construção de software tornou-se centro de muitos projetos. Empresas surgiram especializadas na confecção deste tipo de produto, além da formação de uma comunidade em volta do tema.

Observa-se, então, a tendência e a necessidade de construir software buscando agilidade, encurtando o tempo entre a concepção da ideia, ou surgimento de uma carência a ser resolvida, até a entrega desta solução construída, que ainda que não signifique o fim do ciclo de vida deste produto, pode ser considerado como o marco temporal da implantação.

Logo, este estudo busca apresentar que a arquitetura de microsserviços e funções como serviço é uma opção viável, e juntamente com o formato arquitetural, há sim vários outros benefícios que podem auxiliar a acelerar a construção de soluções. Aliado a isto, a cultura do *DevOps* assegura, em níveis propriamente dispostos, que os sistemas estejam sendo construídos da forma correta, obedecendo a padrões, de forma segura e que estão implantados e em funcionamento no menor tempo hábil. Isto é positivo só não para o usuário final da solução, bem como para o time que desenvolve a mesma.

O desenvolvimento do *Project SRC* foi a tentativa de experimentar a aplicabilidade da solução como um todo, arquitetura, serviços, métodos e tarefas. Servindo ainda, ao final, de base para o projeto externo ao estudo com mesmo nome a ser continuado pelo autor e alguns integrantes da comunidade de automobilismo virtual.

O autor considera que dada a execução do estudo, a quantidade de recursos, resultados e produtos entregues corroboram para reforçar a proposta. Consistindo ainda de uma oportunidade ímpar de estar em contato com diversas tecnologias, aprender com as dificuldades enfrentadas e fazer do relato uma contribuição a todos que estão em contextos similares. Definitivamente, há seções do estudo ficaram aquém do desejado e planejado. Entretanto, estas se apresentam como uma oportunidade para estudos futuros e, irrevogavelmente, serão continuadas no projeto externo (a continuação da execução e construção do Project SRC pelo o autor).

Referências

AHMAD, M. O.; MARKKULA, J.; OIVO, M. Kanban in software development: A systematic literature review. *Proceedings - 39th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2013*, p. 9–16, September 2013. Citado na página 42.

Amazon Web Services, Inc. *What is a Message Queue*. 2019. Disponível em: <https://aws.amazon.com/message-queue/>. Citado 2 vezes nas páginas 37 e 38.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices Architecture Enables DevOps : An Experience Report on Migration to a Cloud-Native Architecture The Architectural Concerns for Microservices Migration The Architecture of Backtory Before the Migration. *IEEE Software*, v. 33, n. 3, p. 42–52, 2016. ISSN 07407459. Citado 2 vezes nas páginas 24 e 39.

BASS, L.; WEBER, I.; ZHU, L. *DevOps: A software architect's perspective*. [S.l.]: Addison-Wesley Professional, 2015. Citado na página 27.

BEIMBORN, D.; MILETZKI, T.; WENZEL, S. Platform as a Service (PaaS). *Business & Information Systems Engineering*, v. 3, n. 6, p. 381–384, dec 2011. ISSN 1867-0202. Disponível em: <http://link.springer.com/10.1007/s12599-011-0183-3>. Citado na página 35.

BHARDWAJ, S.; JAIN, L.; JAIN, S. Cloud Computing: A study of Infrastructure as a Service (IaaS). *International Journal of Engineering and Information Technology*, v. 2, n. 1, p. 60–63, 2010. ISSN 0976-0253. Citado 3 vezes nas páginas 33, 34 e 35.

BURNS, B. et al. Borg, omega, and kubernetes. *ACM Queue*, v. 14, p. 70–93, 2016. Disponível em: <http://queue.acm.org/detail.cfm?id=2898444>. Citado 2 vezes nas páginas 49 e 50.

BUXMANN, P.; HESS, T.; LEHMANN, S. Software as a Service. *WIRTSCHAFTSINFORMATIK*, v. 50, n. 6, p. 500–503, dec 2008. ISSN 0937-6429. Disponível em: <http://link.springer.com/10.1007/s11576-008-0095-0>. Citado na página 36.

Castro, P. et al. Serverless programming (function as a service). In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. [S.l.: s.n.], 2017. p. 2658–2659. Citado na página 37.

CHEN, L. Microservices: Architecting for Continuous Delivery and DevOps. *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*, p. 39–46, March 2018. Citado na página 39.

DAN, M. The business model of "Software-As-A-Service

. *Proceedings - 2007 IEEE International Conference on Services Computing, SCC 2007*, n. July, p. 701–702, 2007. Citado na página 36.

Docker Inc. *Docker: Get Started*. 2019. Disponível em: <https://docs.docker.com/get-started/>. Citado 3 vezes nas páginas 48, 49 e 50.

ELLIS, A. *Introducing Functions as a Service (OpenFaas)*. 2017. Disponível em: <https://blog.alexellis.io/introducing-functions-as-a-service/>. Citado na página 47.

ELLIS, A. *OpenFaas*. 2017. Disponível em: <https://docs.openfaas.com>. Citado na página 47.

ELLIS, A. *OpenFaas Stack*. 2019. Disponível em: <https://docs.openfaas.com/architecture/stack/>. Citado 2 vezes nas páginas 47 e 48.

FARROHA, B. S.; FARROHA, D. L. A framework for managing mission needs, compliance, and trust in the DevOps environment. *Proceedings - IEEE Military Communications Conference MILCOM*, IEEE, p. 288–293, 2014. Citado na página 29.

FOWLER, M.; LEWIS, J. *Microservices: a definition of this new architectural term*. 2015. Disponível em: <https://martinfowler.com/articles/microservices.html>. Citado 2 vezes nas páginas 23 e 30.

FOX, G. C. et al. Report from workshop and panel on the Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research. *Whitepaper*, p. 1–22, 2017. Citado 2 vezes nas páginas 24 e 36.

FYDORENCHYK, T. *Jelastic - SaaS, PaaS, IaaS*. 2019. Disponível em: <https://jelastic.com/blog/turnkey-multi-cloud-paas-freedom/saas-paas-iaas/>. Citado na página 33.

GROHMANN, W. *"Von der Software zum Service": ASP, Software on Demand, Software-as-a-Service, Cloud Computing-neue Formen der Software-Nutzung*. [S.l.]: H. K. P. Consulting, 2009. ISBN 3939968072. Citado na página 35.

GUTHRIE, S. *DevOps Principles- The CAMS Model*. 2019. Disponível em: <https://medium.com/@seanguthrie/devops-principles-the-cams-model-9687591ca37a>. Citado na página 28.

HAYES, B. Cloud Computing. *Communications of the ACM*, v. 51, n. 7, p. 9–11, 2008. ISSN 15577317. Citado na página 32.

HE, H. What is service-oriented architecture. *Publicação eletrônica em*, v. 30, p. 1–5, 2003. Citado na página 23.

JOHNSON, M. *What is a pull request?* 2013. Disponível em: <http://oss-watch.ac.uk/resources/pullrequest>. Citado na página 29.

LEITE, L. et al. A Survey of DevOps Concepts and Challenges. v. 1, n. 1, 2019. Disponível em: <http://arxiv.org/abs/1909.05409>. Citado 2 vezes nas páginas 24 e 27.

MENDES, P. *Microsserviços, por Martin Fowler e James Lewis*. 2016. Disponível em: <http://www.pedromendes.com.br/2016/01/02/microservicos/>. Citado na página 31.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, v. 2014, n. 239, p. 2, 2014. Citado 2 vezes nas páginas 48 e 49.

MYRBAKKEN, H.; COLOMO-PALACIOS, R. DevSecOps: A Multivocal Literature Review. In: *International Conference on ...*. [s.n.], 2017. p. 17–29. ISBN 978-3-319-67383-7. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-319-67383-7{_}3http://link.springer.com/10.1007/978-3-319-67383->. Citado na página 27.

PALKO, T. *Monitoring in the DevOps Pipeline*. 2015. 1 p. Disponível em: <https://insights.sei.cmu.edu/devops/2015/12/monitoring-in-the-devops-pipeline.html>. Citado na página 39.

RADIGAN, D. *Kanban - How the kanban methodology applies to software development*. 2015. Disponível em: <https://www.atlassian.com/agile/kanban>. Citado na página 42.

RedHat Incorporated. *O que é uma API?* RedHat Incorporated, 2019. Disponível em: <https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces>. Citado 2 vezes nas páginas 31 e 32.

RICHARDSON, A. *GitOps - Operations by Pull Request*. 2017. Disponível em: <https://www.weave.works/blog/gitops-operations-by-pull-request>. Citado na página 29.

RILEY, C. *GitOps 101: What Is GitOps, and Why Would You Use It?* 2018. Disponível em: <https://www.twistlock.com/2018/08/06/gitops-101-gitops-use/>. Citado na página 29.

ROUSE, M. *What is NoOps? - Definition from WhatIs.com*. 2015. Disponível em: <https://searchitoperations.techtarget.com/definition/NoOps>. Citado na página 28.

SHARMA, S.; SARKAR, D.; GUPTA, D. Agile Processes and Methodologies: A Conceptual Study. *International Journal on Computer Science and Engineering*, v. 4, n. 05, p. 892–898, 2012. Citado na página 42.

The Linux Foundation. *Kubernetes: What is Kubernetes*. 2019. Disponível em: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>. Citado 3 vezes nas páginas 49, 50 e 51.

TSAI, W. T.; BAI, X. Y.; HUANG, Y. Software-as-a-service (SaaS): Perspectives and challenges. *Science China Information Sciences*, v. 57, n. 5, p. 1–15, 2014. ISSN 1674733X. Citado na página 35.

United Nations. *Internet milestone reached, as more than 50 per cent go online: UN telecoms agency*. 2018. Disponível em: <https://news.un.org/en/story/2018/12/1027991>. Citado na página 23.

VINOSKI, S. Advanced message queuing protocol. *IEEE Internet Computing*, v. 10, n. 6, p. 87–89, November 2006. Copyright - Copyright IEEE Computer Society Nov 2006; Last updated - 2010-06-06; CODEN - IESEDJ. Disponível em: <https://search.proquest.com/docview/197318680?accountid=26646>. Citado na página 37.

WALLER, J.; EHMKE, N. C.; HASSELBRING, W. Including Performance Benchmarks into Continuous Integration to Enable DevOps. *ACM SIGSOFT Software Engineering Notes*, v. 40, n. 2, p. 1–4, 2015. ISSN 01635948. Disponível em: <http://doi.acm.org/10.1145/2735399.2735416http://dl.acm.org/citation.cfm?doid=2735399.2735416>. Citado na página 39.

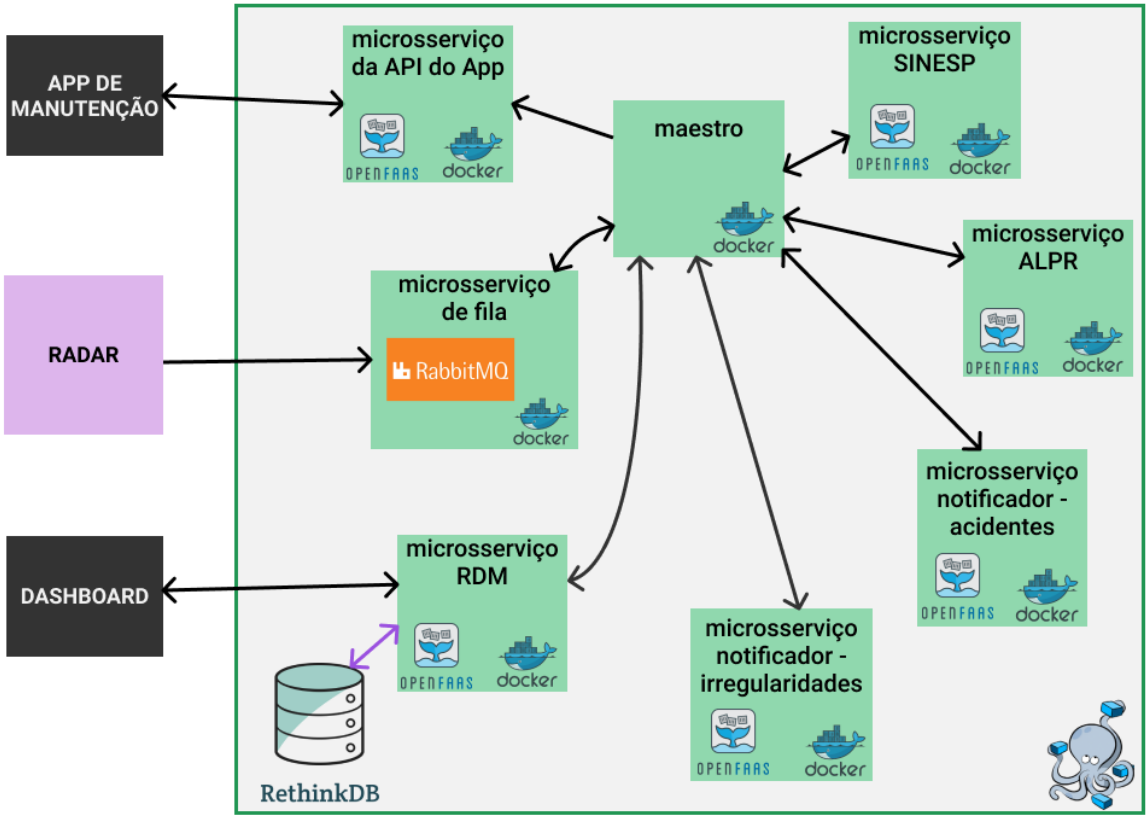
ZHU, L.; BASS, L.; CHAMPLIN-SCHARFF, G. DevOps and Its Practices. *IEEE Software*, IEEE, v. 33, n. 3, p. 32–34, 2016. ISSN 07407459. Citado na página 27.

Anexos

ANEXO A – Diagrama Arquitetural RaDop

O acesso aos sistemas do projeto podem ser feitos na organização do projeto pelo link assinalado. RaDop: <https://github.com/radar-pi>.

Figura 17 – Diagrama de arquitetura de microsserviços do projeto RaDop.



Fonte: Projeto RaDop (2019).

ANEXO B – Documento da Copa SRC - Evento

Copa Project SRC

Copa Projeto SRC, informações para os pilotos. Qualquer dúvida, informação enganosa ou erro, entre em contato com João Pedro Sconetto.

- Abertura de Sala: 22:15 (GMT-3 Horário de Brasília)
- Abertura da Qualificatória: 22:30 (GMT-3 Horário de Brasília)

Configurações Gerais

- **Tipo de Largada**: Largada no grid com verificação de queimada de largada
- **Grid Order**:
    - **Com Qualificatória**: Mais rápido primeiro
    - **Sem Qualificatória**: Grid invertido baseado nos resultados da corrida anterior
- **Turbo**: Forte
- **Força do cone de aspiração**: Real
- **Danos visíveis**: Ligado
- **Danos mecânicos**: Pesado
- **Desgaste dos pneus**: Veja `Pistas`
- **Consumo de combustível**: Veja `Pistas`
- **Combustível inicial**: Tanque Cheio (Padrão)
- **Redução de aderência (Pista molhada/Bordas)**: Real
- **Atraso na chegada**: 30 sec
- **Equlíbrio de desempenho**: Ligado
- **Classificação de pneu máxima**: `Corrida: Suave` & `Corrida: Chuva`
- **Classificação de pneu mínima**: `Corrida: Duro` & `Corrida: Intermediário`
- **Ajuste**: Proibido
- **Uso de Kart**: Desligado
- **Fantasma durante corridas**: Nenhuma
- **Penalidade por atalho**: Forte
- **Penalidade por colisão em mureta**: Penalidade de tempo (severa)
- **Penalidade por contato lateral**: Desligado
- **Corrigir curso do veículo após colisão em mureta**: Desligado

- **Reposicionar carros que saem da pista**: Desligado
- **Regras de bandeira**: Ligado
- **Ativar fantasma para carros com vantagem**: Desligado
- **Assistência de contraesterço**: Proibido
- **Gerenciamento de estabilidade ativo (ASM)**: Proibido
- **Assistência de linha de direção**: Proibido
- **Controle de tração**: Proibido
- **ABS**: Sem limite
- **Piloto automático**: Proibido

Carro

Mazda Roadster Touring Car - N300

- 248 HP/252 CV
- 2207 lbs/1001 kg

- **Observation**: O ED vai configurar automaticamente os valores.

Pistas

1ª Etapa

1. Blue Moon Bay Speedway - Infield A

- Desgaste de Pneus - 10x
- Consumo de Combustível - 10x
- Qualificatória - 5 Min
- Corrida - 15 Min
- Condições Climáticas - 13:00 Tempo Bom

2. Red Bull Ring

- Desgaste de Pneus - 30x
- Consumo de Combustível - 5x
- Corrida - 15 Min
- Condições Climáticas - 17:30 Tempo Aberto
- Grid Inverso

2ª Etapa

3. Autódromo de Interlagos

- Desgaste de Pneus - 10x
- Consumo de Combustível - 10x
- Qualificatória - 5 min
- Corrida - 15 Min
- Condições Climáticas - 08:30 Tempo Aberto

4. Suzuka Circuit

- Desgaste de Pneus - 25x
- Consumo de Combustível - 7x
- Corrida - 20 Min

- Condições Climáticas - 14:30 Tempo Bom
- Grid Inverso

3ª Etapa

5. Sardegna - Road Track - B

- Desgaste de Pneus - 20x
- Consumo de Combustível - 10x
- Qualificatória - 5 Min
- Corrida - 15 Min
- Condições Climáticas - 13:10 Tempo Aberto

6. Tokyo Expressway - East Outer Loop

- Desgaste de Pneus - 15x
- Consumo de Combustível - 3x
- Corrida - 20 Min
- Condições Climáticas - 18:00 Chuva
- Grid Inverso

Pontuação

Estilo FIA GTC

Corridas 1 à 5:

1º - 12 pontos

2º - 10 pontos

3º - 8 pontos

4º - 7 pontos

5º - 6 pontos

6º - 5 pontos

7º - 4 pontos

8º - 3 pontos

9º - 2 pontos

10º - 1 ponto

Última Corrida - Dobro de Pontos:

1º - 24 pontos

2º - 20 pontos

3º - 16 pontos

4º - 14 pontos

5º - 12 pontos

6º - 10 pontos

7º - 8 pontos

8º - 6 pontos

9º - 3 pontos

10º - 2 pontos

ANEXO C – Documento da Copa SRC - Resultados

Copa Project SRC

1º: ***Raokkonen*** - Rafael Almeida de Oliveira

2º: ***Sconetto*** - João Pedro Sconetto

3º: ***Willson_Martins*** - Willson Coelho Martins

Resultados

1ª Etapa

Blue Moon Bay Speedway - Infield A

1. Raokkonen
2. Sconetto
3. Willson_Martins
4. ORMA_Bixigass
5. ORMA_Clemente
6. homem_macacoUSA
7. tirsomed34
8. vyidals
9. pirizao2014
10. THDestro30
11. Valentim87

Red Bull Ring

1. Raokkonen
2. ORMA_Clemente
3. homem_macacoUSA
4. Sconetto
5. ORMA_Bixigass
6. Willson_Martins
7. THDestro30

8. pirizao2014
9. vyidals
10. tirsomed34
11. Valentim87

2ª Etapa

Autódromo de Interlagos

1. Sconetto
2. Raokkonen
3. THDestro30
4. ORMA_Clemente
5. pirizao2014
6. Willson_Martins
7. tirsomed34
8. ORMA_Bixigass

DNF

- homem_macacoUSA
- vyidals
- Valentim87

Suzuka Circuit

1. Raokkonen
2. THDestro30
3. tirsomed34
4. Willson_Martins
5. ORMA_Clemente
6. Sconetto
7. pirizao2014
8. ORMA_Bixigass

DNF

- homem_macacoUSA

- vyidals
- Valentim87

3ª Etapa

Sardegna - Road Track - B

1. THDestro30
2. ORMA_Bixigass
3. Willson_Martins
4. Raokkonen
5. homem_macacoUSA
6. ORMA_Clemente
7. Sconetto
8. tirsomed34

DNF

- vyidals
- Valentim87

Tokyo Expressway - East Outer Loop

1. Sconetto
2. tirsomed34
3. homem_macacoUSA
4. Willson_Martins
5. Raokkonen
6. ORMA_Clemente
7. ELITE_Caca93
8. ORMA_Bixigass

DNF

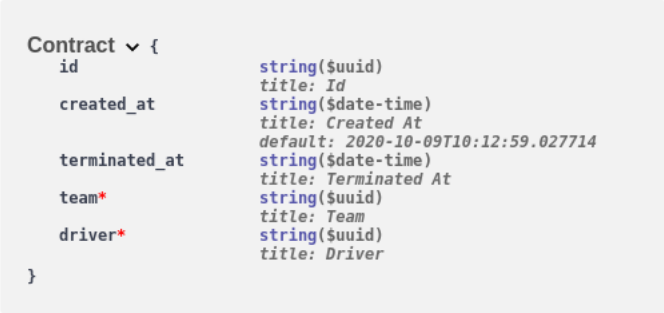
- THDestro30
- vyidals
- Valentim87

Tabela de Pontuação

Posição	Piloto	Pontuação
1º	Raokkonen	65 pontos
2º	Sconetto	62 pontos
3º	Willson_Martins	47 pontos
4º	ORMA_Clemente	44 pontos
5º	tirsomed34	40 pontos
6º	ORMA_Bixigass	37 pontos
7º	homem_macacoUSA	35 pontos
8º	THDestro30	35 pontos
9º	pirizao2014	15 pontos
10º	vyidals	5 pontos
11º	Valentim87	0 pontos

ANEXO D – Modelo de Dados *Project SRC* - Atributos das Entidades

Figura 18 – Atributos da entidade *Contract*.

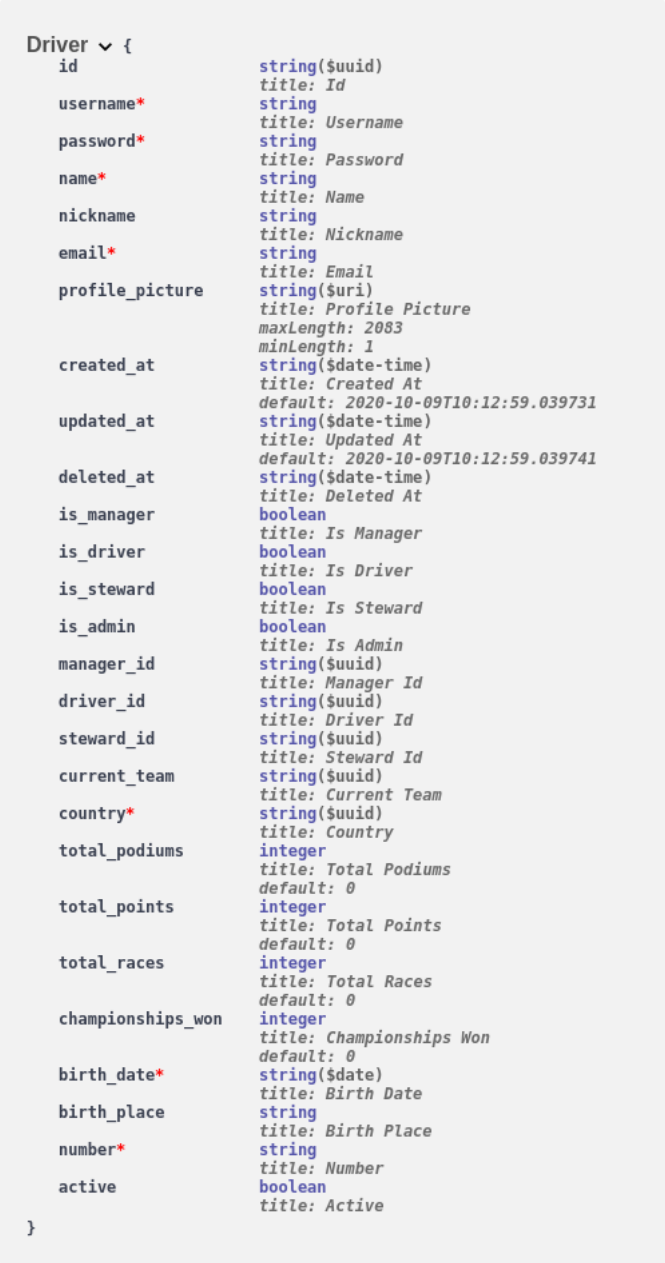


Fonte: João Pedro Sconetto (2020).

Figura 19 – Atributos da entidade *Country*.

Fonte: João Pedro Sconetto (2020).

Figura 20 – Atributos da entidade *Driver*.



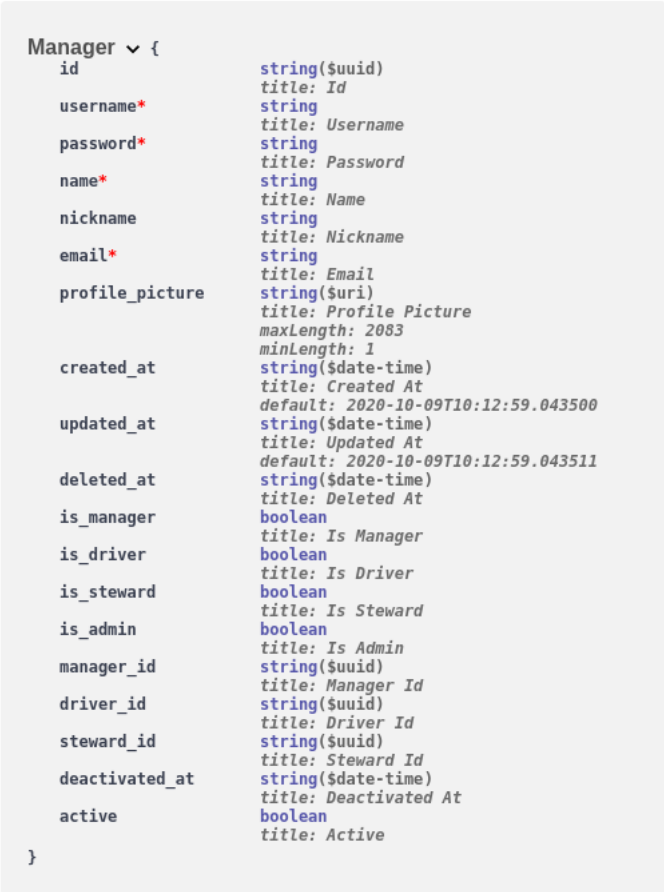
```
Driver ⌄ {
  id                  string($uuid)
                      title: Id
  username*           string
                      title: Username
  password*           string
                      title: Password
  name*               string
                      title: Name
  nickname            string
                      title: Nickname
  email*              string
                      title: Email
  profile_picture     string($uri)
                      title: Profile Picture
                      maxLength: 2083
                      minLength: 1
  created_at          string($date-time)
                      title: Created At
                      default: 2020-10-09T10:12:59.039731
  updated_at          string($date-time)
                      title: Updated At
                      default: 2020-10-09T10:12:59.039741
  deleted_at          string($date-time)
                      title: Deleted At
  is_manager          boolean
                      title: Is Manager
  is_driver           boolean
                      title: Is Driver
  is_steward          boolean
                      title: Is Steward
  is_admin            boolean
                      title: Is Admin
  manager_id          string($uuid)
                      title: Manager Id
  driver_id           string($uuid)
                      title: Driver Id
  steward_id          string($uuid)
                      title: Steward Id
  current_team        string($uuid)
                      title: Current Team
  country*            string($uuid)
                      title: Country
  total_podiums       integer
                      title: Total Podiums
                      default: 0
  total_points        integer
                      title: Total Points
                      default: 0
  total_races         integer
                      title: Total Races
                      default: 0
  championships_won   integer
                      title: Championships Won
                      default: 0
  birth_date*         string($date)
                      title: Birth Date
  birth_place         string
                      title: Birth Place
  number*             string
                      title: Number
  active              boolean
                      title: Active
}
```

Fonte: João Pedro Sconetto (2020).

Figura 21 – Atributos da entidade *League*.

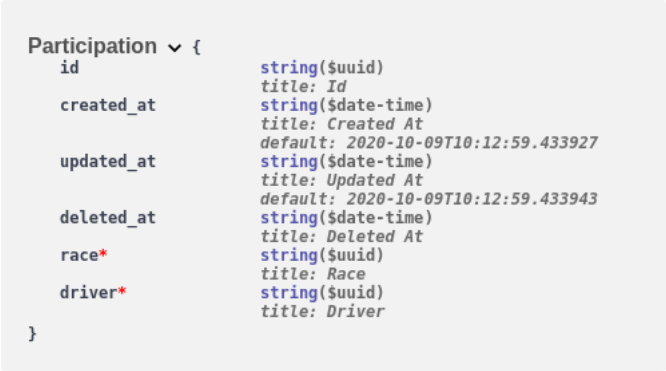
Fonte: João Pedro Sconetto (2020).

Figura 22 – Atributos da entidade *Manager*.



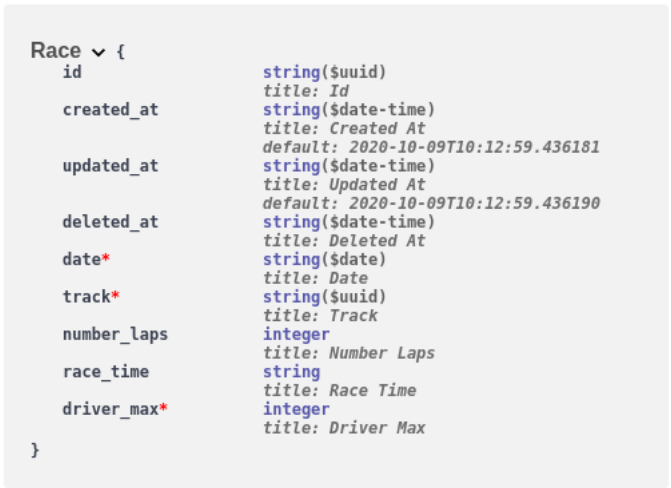
Fonte: João Pedro Sconetto (2020).

Figura 23 – Atributos da entidade *Participation*.



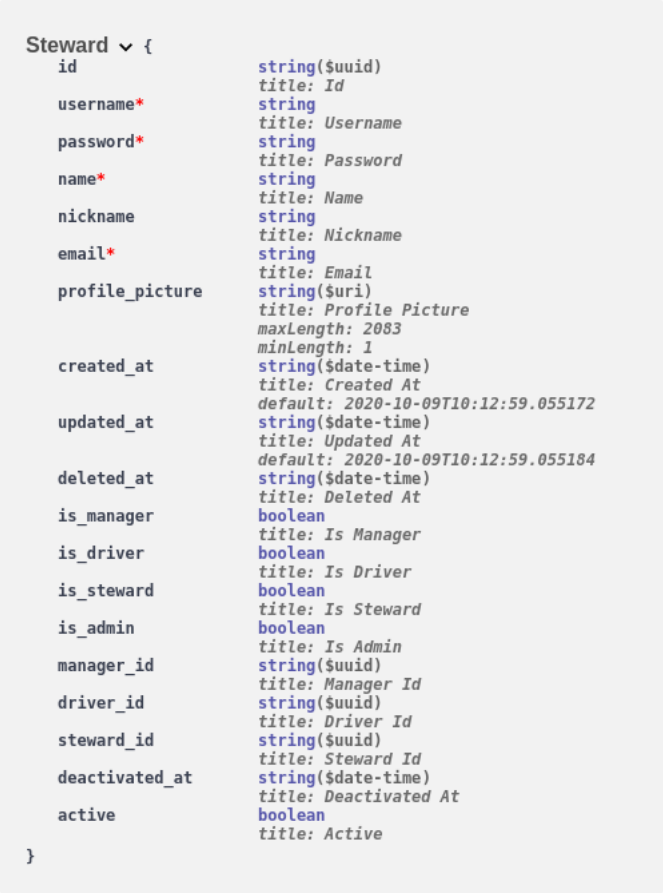
Fonte: João Pedro Sconetto (2020).

Figura 24 – Atributos da entidade *Race*.



Fonte: João Pedro Sconetto (2020).

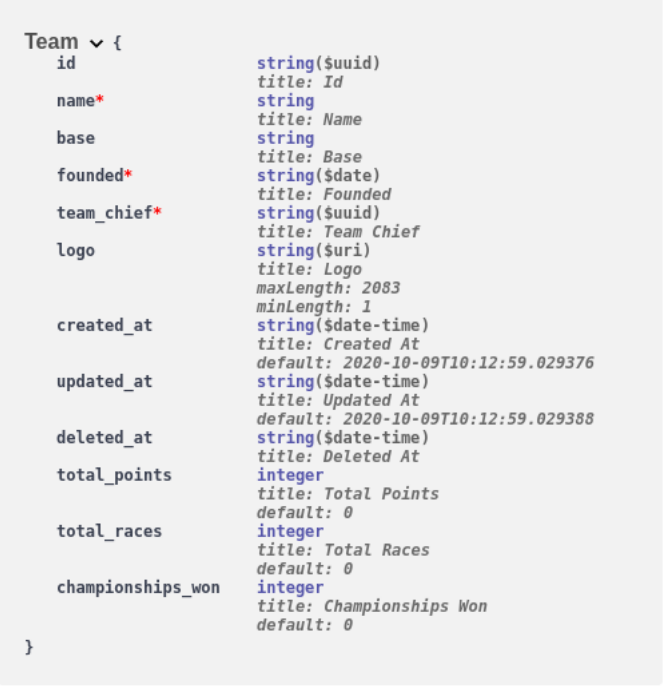
Figura 25 – Atributos da entidade *Steward*.



```
Steward ⌄ {
  id                  string($uuid)
                      title: Id
  username*           string
                      title: Username
  password*           string
                      title: Password
  name*               string
                      title: Name
  nickname            string
                      title: Nickname
  email*              string
                      title: Email
  profile_picture     string($uri)
                      title: Profile Picture
                      maxLength: 2083
                      minLength: 1
  created_at          string($date-time)
                      title: Created At
                      default: 2020-10-09T10:12:59.055172
  updated_at          string($date-time)
                      title: Updated At
                      default: 2020-10-09T10:12:59.055184
  deleted_at          string($date-time)
                      title: Deleted At
  is_manager          boolean
                      title: Is Manager
  is_driver           boolean
                      title: Is Driver
  is_steward          boolean
                      title: Is Steward
  is_admin            boolean
                      title: Is Admin
  manager_id          string($uuid)
                      title: Manager Id
  driver_id           string($uuid)
                      title: Driver Id
  steward_id          string($uuid)
                      title: Steward Id
  deactivated_at      string($date-time)
                      title: Deactivated At
  active              boolean
                      title: Active
}
```

Fonte: João Pedro Sconetto (2020).

Figura 26 – Atributos da entidade *Team*.



```
Team ⌄ {
  id                  string($uuid)
                      title: Id
  name*               string
                      title: Name
  base                string
                      title: Base
  founded*            string($date)
                      title: Founded
  team_chief*         string($uuid)
                      title: Team Chief
  logo                string($uri)
                      title: Logo
                      maxLength: 2083
                      minLength: 1
  created_at          string($date-time)
                      title: Created At
                      default: 2020-10-09T10:12:59.029376
  updated_at          string($date-time)
                      title: Updated At
                      default: 2020-10-09T10:12:59.029388
  deleted_at          string($date-time)
                      title: Deleted At
  total_points        integer
                      title: Total Points
                      default: 0
  total_races         integer
                      title: Total Races
                      default: 0
  championships_won   integer
                      title: Championships Won
                      default: 0
}
```

Fonte: João Pedro Sconetto (2020).

Figura 27 – Atributos da entidade *Track*.

Fonte: João Pedro Sconetto (2020).